SYBEX

THE EASY GUIDE TO YOUR APPLE II®

JOSEPH KASCMER

THE EASY GUIDE TO YOUR APPLE II®

THE EASY GUIDE TO YOUR APPLE II®

JOSEPH KASCMER

Berkeley • Paris • Düsseldorf

Cover art by Sato Yamamoto

Apple II, II + , IIe, Applesoft, and Integer BASIC are registered trademarks of the Apple Computer Corporation

Every effort has been made to supply complete and accurate information. However, Sybex assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result.

Copyright ©1983 SYBEX Inc. World Rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 83-60949
ISBN 0-89588-122-5
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

To Agnes Franz,
and all who bring into the future the best of the past.

CONTENTS

Acknowledgements viii
Preface ix

1 *TAKING CONTROL* 1

Master and Servant Meet 1
Setting Up the Computer 2
Control from the Keyboard 4

2 *VIDEO TYPING AND DRAWING* 9

Screen Control 9
Video Printing 12
Commands in Groups 18
Mosaics—The 40-by-40 Graphics Mode 20
Sketches—The 280-by-160 Graphics Mode 23

3 *SECRET SCHEMES—PROGRAMS* 27

Setting a Schedule 27
Stand-Ins For Numbers 33
Making Your Computer Decisive 37

4 *ALL MANNER OF MANIPULATIONS* 43

A Programmer's Kit of Commands 43
Commands for Moving Among Statements 50
Repeating Within Limits 54
Words from the Computer 61
Inside Information 68
Outside Contact 75

5 *MOBILE MEMORIES—DISKS* 83

A Marriage of Machines 83
Using Prerecorded Disks 88
Programs from Computer to Disk 97
Facts and Files 103

6 *CONTROL COMBINATIONS* 111

Adding Program Parts 111
Mingling Disk Programs 117

7 *THE INSIDE STORY* 123

A Step Past the Translator 123
What Are the Apple II's? 128
Expanding the Computer 132

APPENDICES 135

A Special-Purpose Commands 135
B Jargon Phrase Guide 138
C Books of Interest 143

Index 145

ACKNOWLEDGEMENTS

The people who made this book include: James Compton, who applied his experienced hand as editor to smooth and polish its rough edges; Rudolph Langer, who provided the idea from which it germinated; Ingrid Owen and Margaret Cusick, who added the spark of their artwork; Donna Scanlon and Elaine Foster, who each helped my penned changes find their way into the word processor; Joel Kreisman, who attended to technical details; and Dirk van Nouhuys at Apple Computer, Inc., who thoughtfully provided information.

The two lines of poetry on page 10 are from "Earth's Story," by Thomas Curtis Clark. The poem, "The Sea Gypsy," on page 11 and 13, was written by Richard Hovey.

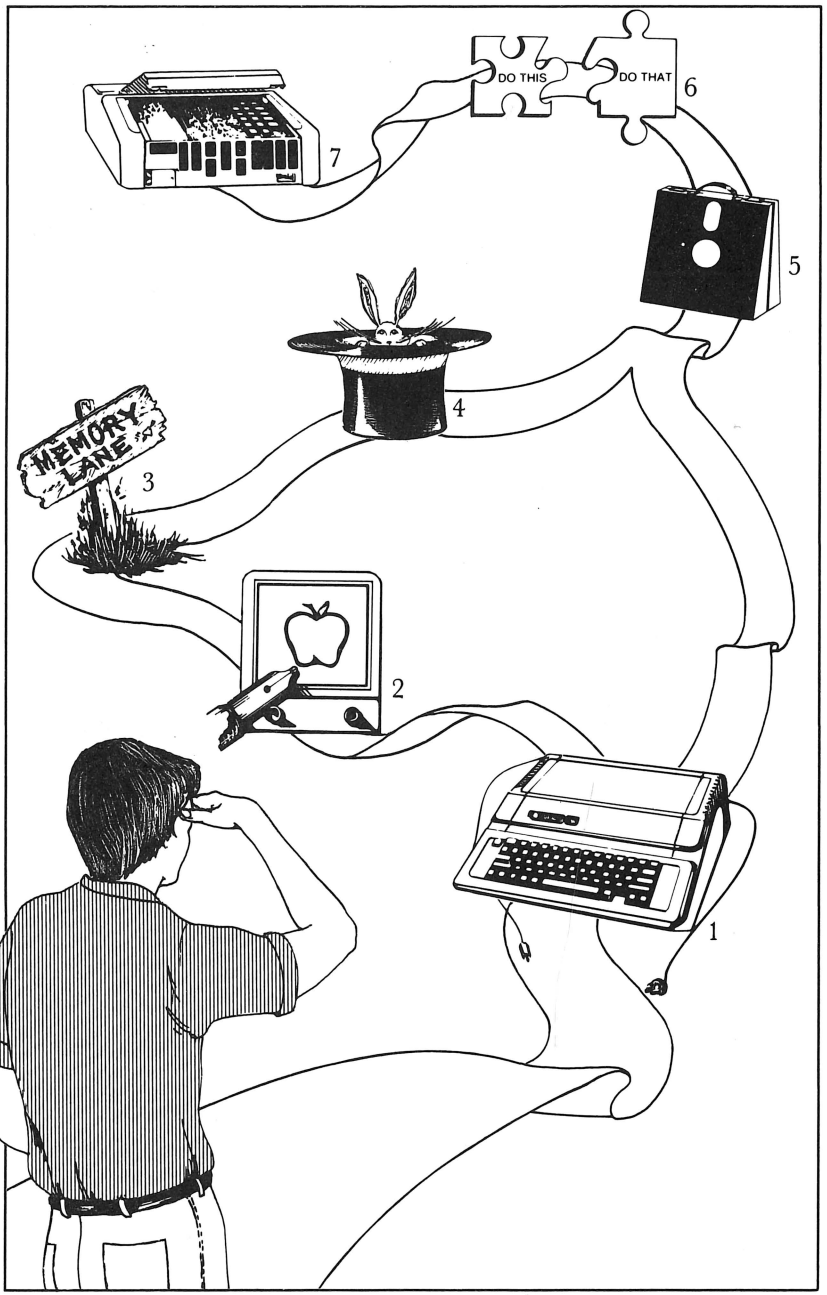
J.K.

PREFACE

This book is about controlling your computer. In it, you will see how mastery over the most versatile of the "people's computers" can be achieved in a few days, without the burden of jargon or theories of computer science and construction. The half-million Apple II computers in use, the thousands of prerecorded programs on disk, and the scores of plug-in circuits and devices made for them all testify to the widespread usefulness of the machine. The most recent and refined version of the Apple II design, the IIe version featured in the descriptions herein, is a tool useful to computer newcomer and enthusiast alike.

In explaining the use of the Apple II, this book is really about you, the user, and the ways you can direct a computer to work as an extension of your own mind. You need no bent for programming, or even mathematics, to gain the control explained in this book, just as you need no veterinary skills to command a dog or no mechanical aptitude to drive a car. You need only a desire to extend your mental processes through a machine different from, but no more difficult to operate than, an automobile or a typewriter. If, in short, you have use for the Apple II, or if you are interested in finding out how you can control this type of computer, then read on, uninhibited, for this is your book.

Written to cover the IIe version, this book—with a few references to the keyboard excepted—will serve well as a guide to the immediate predecessor, the II + version, also. The earliest version of the machine, however, known simply as the Apple II, differs so substantially from its successors that some of the examples given will not fully apply, although many of the descriptions will. Throughout this book, the name "Apple II" is used generically, to refer to the *model* and not to any particular *version* of it.



Two ways of going through this book: for disk use, or for programming also.

Because you can use a computer in more than one way, you can approach this book from more than one direction.

Chapter 1 is a brief introduction telling where the controls are, how to plug together your computer system, and what buttons to push and not to push. No matter what you plan to use your computer for, a run through this chapter will put you at ease with the box in front of you.

If you plan to use only prerecorded disks and wish to avoid writing computer directions yourself, you can skip ahead to **Chapter 5.** And if you'd like to make full use of the programs on disks, read the second part of **Chapter 6.**

If you'd like to write your own directions to the computer, you should read **Chapters 2, 3, and 4,** and also the first part of **Chapter 6.**

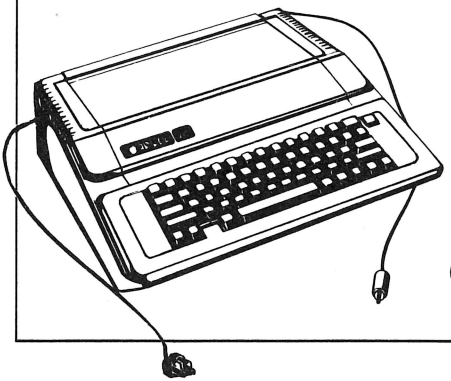
Chapter 7 is the most optional of this book. If you'd like more control over the Apple than provided through the BASIC commands of the IIe or II +, or have an interest in how the computer works, and what else can be done with a more customized computer, read Chapter 7. It will help you understand the machine and expanding your abilities with it. For using the Apple II, though, it's not essential.

The chapters concentrate on commands most useful to the general reader. You'll find other available commands built into the Apple II listed in **Appendix A** according to their type and use.

In the interest of clarity and ease of comprehension, jargon has been avoided in the chapters that follow; however, you may encounter other computer users, salespersons or books lapsing into it. **Appendix B** can help make them intelligible to you.

Every effort has been made to make this book the most useful and comprehensible introduction to an exceptional class of computers. Other books can take you further into the complexities of the machine itself or outward into other uses. You'll find some of these in **Appendix C.**

Finally, this book has been written in, and inspired by, the belief that as personal computers approach widespread use, they can be, more so than the automobile or the telephone, a liberating and expansive experience for the intelligent user, thoughtfully guided. To this end, the following pages are offered in the hope that the time and use to which you put your computer may prove personally enriching.



CHAPTER ONE

TAKING CONTROL

MASTER AND SERVANT MEET

To many people, the computer is a fearsome thing. Working in secret, with a memory nearly unfailing and a speed unimaginable, it's often preceded by an awe and mystery that overwhelm the newcomer.

People have controlled creatures of special abilities for thousands of years. Oxen were tamed as beasts of burden, dogs as guards and companions. In each case the master achieved control through commands. You can command any model of the Apple II computer with a command vocabulary comparable in size to that of a highly trained canine.

A computer under your command can choose among alternatives, organize information, search for facts and patterns, and control devices. It can be prepared to begin any duty by the command of a single word or by the triggering of a single event.

No computer is all-powerful or all-knowing, of course, and there are limits to the operation of any machine. But those limits arise from the design, the speed, and the size of the computer's memory and the availability of connectable circuits and prepared programs of commands. In its class, the Apple IIe ranks as one of the most advanced computers in the world. Plug-in circuits available for it number in the scores, packaged programs in the thousands.

You can gain control of the Apple II in a short time. But to command any computer, you must first become its master. This book will show you a clear way through that transition.

SETTING UP THE COMPUTER

The undemanding Apple II can be set up by anyone who can position a typewriter and hook up a television. It sits and works well anywhere a typewriter might, in any place undisturbed by strong vibrations, extremes of temperature, dust, or humidity. The Apple II will work well most places its operator is comfortable. A typical setup is shown in Figure 1.1.

Figure 1.1: Typical arrangement of Apple II and video monitor.

When firmly seated on a flat surface, the case is strong enough to support a large portable television set or video monitor (a television without a channel tuner). If you don't want it there, you can locate the television or monitor anywhere reached by the cable from the computer.

By connecting only two electrical cords, one for power and one for the television or monitor, you can prepare the Apple II for use. One end of each cord plugs into the back of the computer. If you look at the back panel of the Apple II as shown in Figure 1.2, you'll see a switch with the markings, ON and OFF, and a boxed recess, from which project three metal posts. Into that recess you can plug the boxy end of the power cord. With the switch turned to OFF, you can plug the three-pronged end of the power cord into a wall outlet to complete the power connection.

The connection to a monitor is just as simple. Across the back panel from the power switch, a row of metal jacks faces backward. The jack nearest the edge of the panel takes the video monitor cable. If you're connecting a video monitor, you can use the monitor's video cable, on one end of which is a heavy threaded metal connector, and on the other end a metal plug. You can fit the plug into the computer's video jack. The other end plugs into the threaded jack of the monitor, on which small nubs align with the v-notches of the connector, around which the sleeve screws the connection tight.

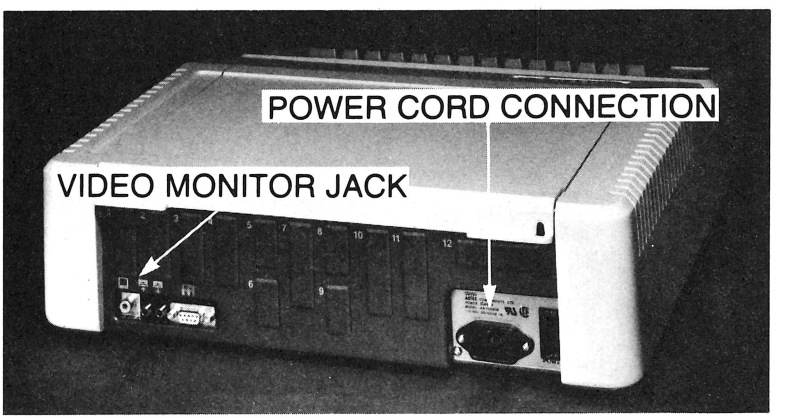


Figure 1.2: Back panel connections to power and video monitor.

If you use a television, you can connect a specially made adaptor, called an RF modulator, inside the Apple II case. The directions provided with the modulator should explain the connection. You can open the Apple's case by pulling up the lid from the back corners, as shown in Figure 1.3. As you pull up, each corner should snap open. Then you can take off the cover by lifting the rear end to about twice the height of the case, and sliding it back. You can replace the cover by slipping the front end below the keyboard, and pressing the back corners until they snap in to lock. With the power cord and video cord thus connected, you have before you a fully operational computer that can display what it does on a screen.

CONTROL FROM THE KEYBOARD

In front of you sits all you need to command the computer. The keyboard is laid out much like a typewriter's, with a few extra keys. Before turning on power to the computer, consider a simple, comforting fact: Nothing you can do from the keyboard will damage the computer. With some of the commands you'll find in this book, you can change the way the computer works, the screen display, and the active memory, but when you turn power to the Apple II off and then on again, you wipe clean the slate of changes. The computer then greets you in readiness, any keyboard manipulations forgiven and forgotten.

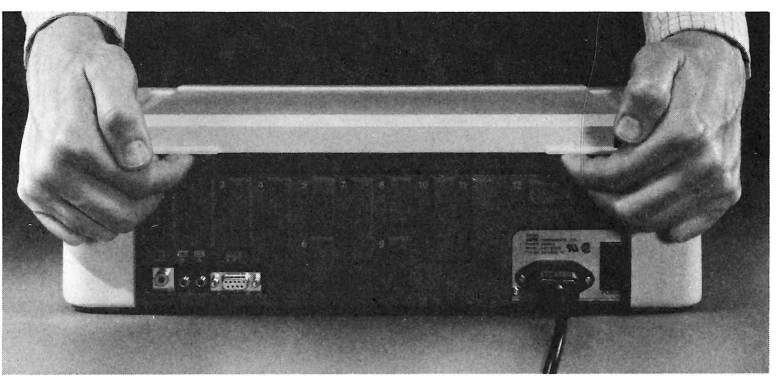


Figure 1.3: Opening the case by pulling up on the lid.

The single most useful piece of advice in the early stages, while you're becoming familiar with the keyboard operation of the computer, then, is this: If something interests you, try it. The worst calamity to befall you will be the loss of some command, information or calculation. In the earliest stages, while you're experimenting, this loss is part of learning to control the beast. At any point, the computer will emerge from human keyboard error or curiosity completely unscathed.

When you switch on first the monitor or television, then the computer, a beep will sound, and printing will appear on the screen. Across the top, if all is connected and working, "APPLE][" will be printed as light letters on a dark screen, as in Figure 1.4. At the upper left of the screen you will see a right bracket mark,], and next to that a single, character-sized flashing square, a signal from the computer that it's ready to read whatever you type from the keyboard. That flashing square marks where the next character from the keyboard will appear on the screen; it's known as the *cursor.*

Designed to display immediately what's typed from the keyboard, the Apple II will print light letters on the dark screen as you

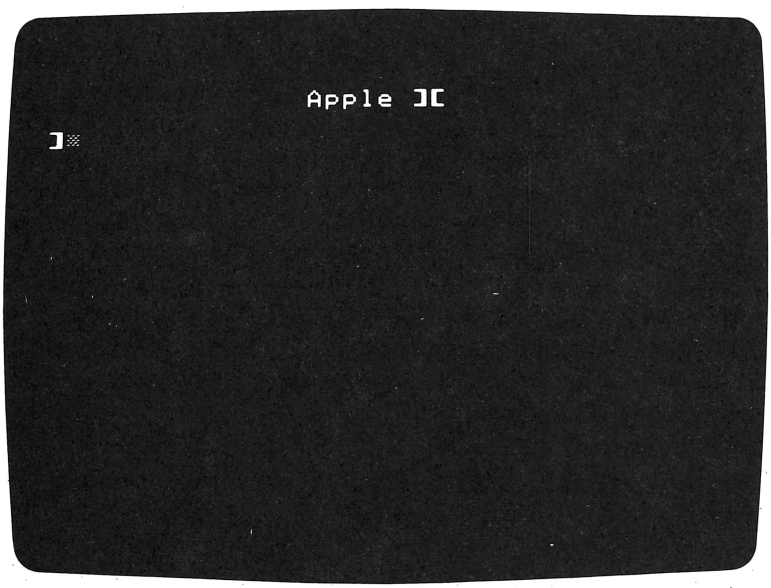


Figure 1.4: The screen display after the Apple II is first turned on.

type them. The right bracket mark,], is a signal from the computer that it's ready to read a command. But whether you type one of the hundred commands the Apple II can act on or Japanese poetry, your typing will appear on the screen. As you type, the computer waits for a signal from you that will tell it to act on the command you've typed between the right bracket mark and the current location of the flashing cursor.

You can signal the computer to act on a typed command by pressing the RETURN key. But if you merrily type along without pressing it, the computer will follow its built-in directions and wait for no more than 256 characters of typing after the right bracket. Then the computer prints another right bracket, follows your typing and waits again for a press of the RETURN key. As a signal to you that it's about to pass out of whatever statement you're typing, the Apple II will sound a beep as you type each of the last eight characters of the 256. Then a left-leaning slash mark (\ ) appears as a signal that all the characters typed after the last right bracket will no longer be considered for action.

You can use the CAPS LOCK key on the IIe version like the shift lock on most typewriters, with the difference that only letter keys are shifted to upper case. Number- and punctuation-key use is unaffected by the CAPS LOCK key. You can use the SHIFT key to type uppermost characters on any key, though.

Treating the screen of the Apple II like a sheet of paper in a typewriter, you can type on it anywhere within its margins. Using the four arrow keys on the IIe, you can move the flashing cursor to any location on the screen, and there type from the keyboard. The left arrow acts as a backspace key. The right arrow advances the cursor along a line of type. The down arrow drops the cursor. Once the cursor is at the bottom of the screen, the down arrow cranks up all the typed lines like the printing on an endless roll of paper being fed through a typewriter.

Although changes are not immediately apparent on the screen, the arrow keys can remove or return screen typing to the computer's memory. The left arrow sends the cursor back over typed characters and deletes them from the computer's memory. The right arrow re-inserts those characters back into memory as it sends the cursor over them.

When switched into action by the key marked ESC, the up arrow key can move the cursor toward the top line of the screen.

You can use the arrow keys to move the cursor without affecting what's already on the screen, by pressing the ESC key. Pressed once, this key switches the arrow keys into a kind of free-ranging mode in which the cursor passes over screen typing without affecting computer memory. Pressed a second time, the ESC key switches the arrow keys back into their more limited mode of operation, in which the up arrow doesn't work and the left and right arrow keys delete and insert into memory those characters over which the cursor is sent.

Two of the keys on the Apple IIe work only when special directions are put into the computer. These are the TAB and DELETE keys, which might prove useful in word processing arrangements.

Both the keys marked ESC and CONTROL can be used with other keys for special purposes. They each do nothing themselves, but allow the other keys to be used in various ways.

As a signal to the computer to act on what you've typed at the keyboard, the RETURN key works as your messenger in sending keyboard statements into the translator of the Apple II. When you press the RETURN key after typing something following one of the right brackets, the computer's translator compares what you've sent it with the vocabulary of words built into it. If the form of your typed statement matches, the translator passes the orders onto the processor part of the computer, which takes action according to built-in directions.

You can move the cursor anywhere around the screen and type anything you like. If you press the RETURN key and the translator doesn't match that with its vocabulary, the computer will signal you that there is a mismatch. In the Apple II, this signal takes the form, "?SYNTAX ERROR". This type of "ERROR" reply from the computer is nothing more than a momentary lapse in communication. In any case, the RETURN key is one you'll find useful when you're ready to give commands to the computer.

Besides the special-character keys (like the foreign-language tilde and accent mark found next to the CAPS LOCK key) two other keys, special-purpose in function, appear on the IIe keyboard on either side of the space bar. To the left is the hollow outline of an apple. You can use this key with two others, the key marked CONTROL and the one marked RESET, to effectively turn the computer on and off from the keyboard without actually losing the power. When, by a bit of finger stretching, you press all

Figure 1.5: The keyboard of the Apple IIe.

three keys at once, first the active part of the computer's memory is wiped clean by an internal loss of power, then power is restored to the Apple IIe circuits.

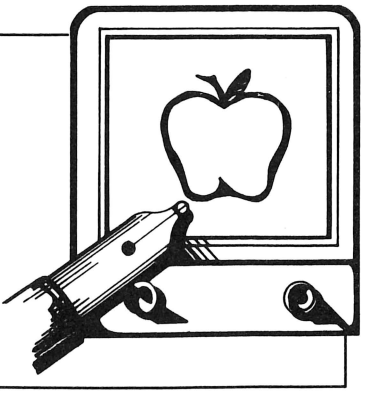
The other special-purpose key on the IIe, a "solid apple," sits to the right of the space bar. You can use it to perform a veiled check of the computer's circuits. Pressing the solid-apple key together with the CONTROL and RESET keys directs the computer to check its own circuits. Patterns will appear on the screen and finally the words, "KERNEL OK," if the computer passes its own self-test. You might try this test after you first set up the Apple IIe, but you will probably find little use for it again.

Each key has an effect (prints a character or sends a command), and some have several different effects when used in combination. You can acquire programs of commands that will change the effect of any or all of the keys. Pressing a key simply provides a signal to the computer, which responds to that signal by printing a character or taking some preplanned action. All the keys are shown in Figure 1.5.

Whether you are using prerecorded commands (on disks, as explained in Chapter 5) or direct commands, you ultimately control the Apple II through the keyboard. The keyboard serves as reins and bridle to the computer and the devices attached to it. And by using the commands explained in the chapters that follow, you can guide the tamed beast in one direction or another, and even send it galloping off to do your bidding on its own.

CHAPTER TWO

VIDEO TYPING AND DRAWING



SCREEN CONTROL

No matter what your computer does internally, it can only communicate with you through the devices to which it's connected. In the simplest arrangement, that's the video screen. And there appear the responses of your electronic servant.

Of course, simply relaying your keyboard character-strikes to the video screen draws upon none of the Apple II's real power. You can direct the computer's power with commands matching those to which it's been built to respond. To some commands, the computer reacts instantly when you send them by pressing the RETURN key. Among these are the commands with which you can fashion the screen display.

Typing on and on from the keyboard without sending the computer a recognizable command, you can fill the screen with any typing. As you type more than a screenful, the computer moves the entire display up one line to make room for the most recent line and loses the topmost line from the screen.

With a one-word command, HOME, you can direct the computer to clear the screen of all printing and to move the flashing cursor to the upper left corner. When so commanded, the computer gives a clean slate, regardless of what else was typed onto the screen. If you wax poetic, for instance, and type onto the screen something like that in Figure 2.1, you can remove it at once.

Directed by command, the computer presents a blank screen

with a right bracket mark and flashing cursor at the upper-left corner—the video equivalents of a fresh sheet of paper and poised printing pen.

You can send this command (as you can any other) by typing it after a right-bracket mark, then pressing the RETURN key. If you typed the text above, and now press RETURN, you will be answered with "? SYNTAX ERROR" and a fresh right-bracket mark. Now you can direct the clearing of the screen and positioning of the cursor by sending the command:

HOME

On receiving this command (sent by the RETURN key), the computer's translator relays it to the processor, which blanks the screen and prints a flashing cursor next to a fresh right bracket mark. The screen is once again at your disposal.

Each time you send the computer the command HOME, with a press of the RETURN key, the screen will appear blanked and ready.

As you type keyboard characters, the computer sends them to the screen as white dot formations on a darkened background.

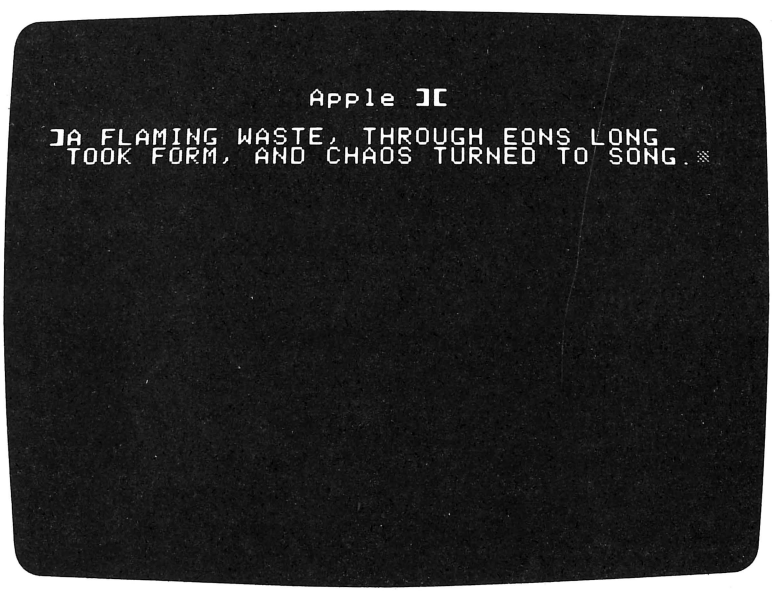


Figure 2.1: One possible screen display.

Likewise, it answers you that way, in a white-on-dark display it chooses automatically. And though white-on-dark characters are probably the easiest on the eyes, there are two alternative ways of putting text on the screen, useful in varying the display or calling attention to particular parts of text.

If you want dark characters to appear, each on their own white background, you can direct the IIe computer to print them that way with the command:

INVERSE

and everything that follows on the screen will be printed in what is called "inverse video." This command controls the computer like a switch, so that until some other command is sent to switch the screen display differently, all printing on the screen will appear as dark-on-white characters. Characters typed before the INVERSE command is sent remain white-on-dark, the "normal" display the computer always presents automatically, until otherwise directed. A long-gone poet, giving the INVERSE command at the beginning of his verse, would have seen a screen like that in Figure 2.2.

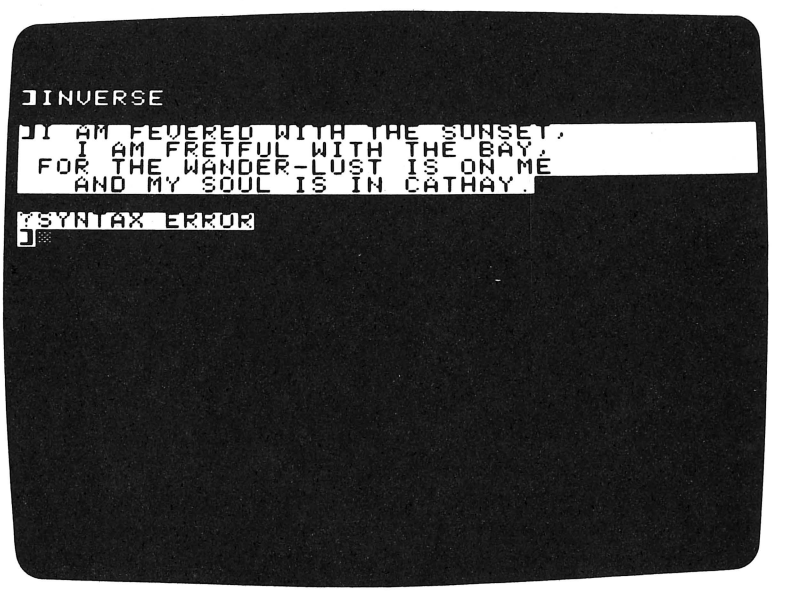

]INVERSE
]I AM FEVERED WITH THE SUNSET,
I AM FRETFUL WITH THE BAY,
FOR THE WANDER-LUST IS ON ME
AND MY SOUL IS IN CATHAY.
?SYNTAX ERROR
]

Figure 2.2: The effect of the INVERSE command.

You can switch the computer back to the white-on-dark mode with the command:

```
NORMAL
```

which directs the computer to print white-on-dark everything typed after the command is sent by the RETURN key. The same poet, advancing through his poem, who then sends the command NORMAL, would see the display of Figure 2.3.

In a third printing mode, the computer will FLASH characters between the white-on-dark NORMAL display and the dark-on-white INVERSE display after a command is sent. After using this command, (FLASH, not surprisingly) the poet would find his text on a screen like that shown in Figure 2.4.

You can switch among any of the three text printing displays with the NORMAL, INVERSE, and FLASH commands.

A word about the IIe version: Although lower-case printing is possible by releasing the CAPS LOCK key on this machine, when it is switched into INVERSE or FLASH modes, the computer—quirky in this respect—will display, instead of lowercase letters, a variety of symbols.

VIDEO PRINTING

The Apple II, like any computer worth its circuits, can do much more than display your keyboard strokes on a video screen. In fact, much of the Apple II's computing power lies in the built-in operations it works on furiously at your command, without giving so much as a flicker of a clue as to what it's doing.

The simplest example can be seen with elementary arithmetic, which the Apple II, like any computer responding to a useful computer language, works on automatically and unseen when directed. In the Apple II the familiar calculator functions of addition, subtraction, multiplication and division are commanded from the keyboard with the symbols +, -, *, and /. So, if you direct the computer to work out the multiplication of -2 and 4 with the command:

```
-2*4
```

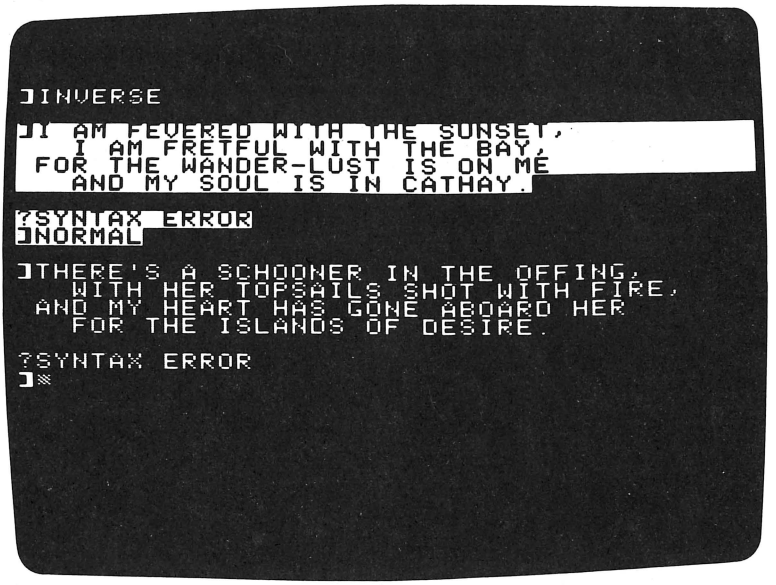

]INVERSE
]I AM FEVERED WITH THE SUNSET,
I AM FRETFUL WITH THE BAY,
FOR THE WANDER-LUST IS ON ME
AND MY SOUL IS IN CATHAY.
?SYNTAX ERROR
]NORMAL
]THERE'S A SCHOONER IN THE OFFING,
WITH HER TOPSAILS SHOT WITH FIRE,
AND MY HEART HAS GONE ABOARD HER
FOR THE ISLANDS OF DESIRE.
?SYNTAX ERROR
]

Figure 2.3: The effect of the NORMAL command.

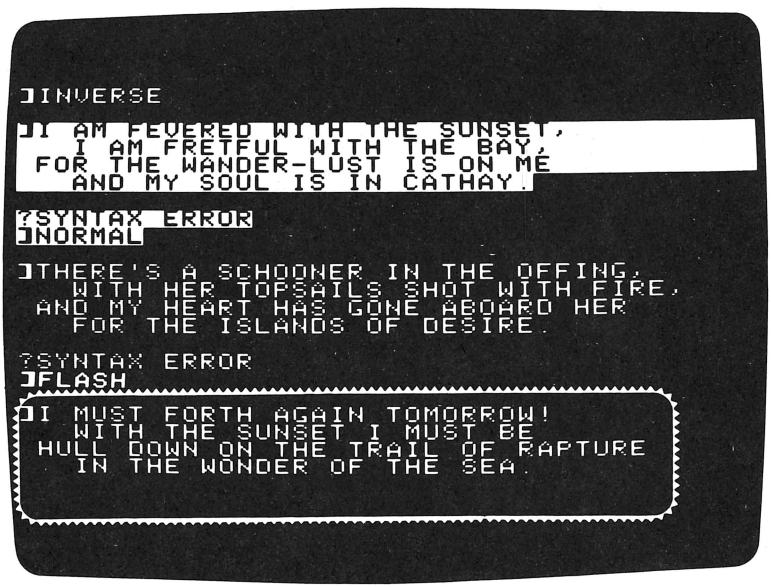

]INVERSE
]I AM FEVERED WITH THE SUNSET,
I AM FRETFUL WITH THE BAY,
FOR THE WANDER-LUST IS ON ME
AND MY SOUL IS IN CATHAY.
?SYNTAX ERROR
]NORMAL
]THERE'S A SCHOONER IN THE OFFING,
WITH HER TOPSAILS SHOT WITH FIRE,
AND MY HEART HAS GONE ABOARD HER
FOR THE ISLANDS OF DESIRE.
?SYNTAX ERROR
]FLASH
]I MUST FORTH AGAIN TOMORROW!
WITH THE SUNSET I MUST BE
HULL DOWN ON THE TRAIL OF RAPTURE
IN THE WONDER OF THE SEA.

Figure 2.4: The effect of the FLASH command.

you won't see the result, just a ?SYNTAX ERROR reply and another right bracket. If you'd like to see the results of that multiplication you can so command it.

The command that directs the computer to display results on the screen is PRINT. It can be used with a command like multiply, *, to display an internal operation of the computer. You can type this useful command first, before any operation you'd like displayed, like this:

```
PRINT -2*4
```

As used here, the PRINT command directs the computer to display on the screen the result of the multiplication commanded after it. *After sending this command with a press of the RETURN key,* you'd see the value −8 printed, and a new right bracket prompt below that, signalling the computer's readiness for another command, as in Figure 2.5.

You can put the PRINT command into service for any combination of arithmetic commands according to the rules of algebra,

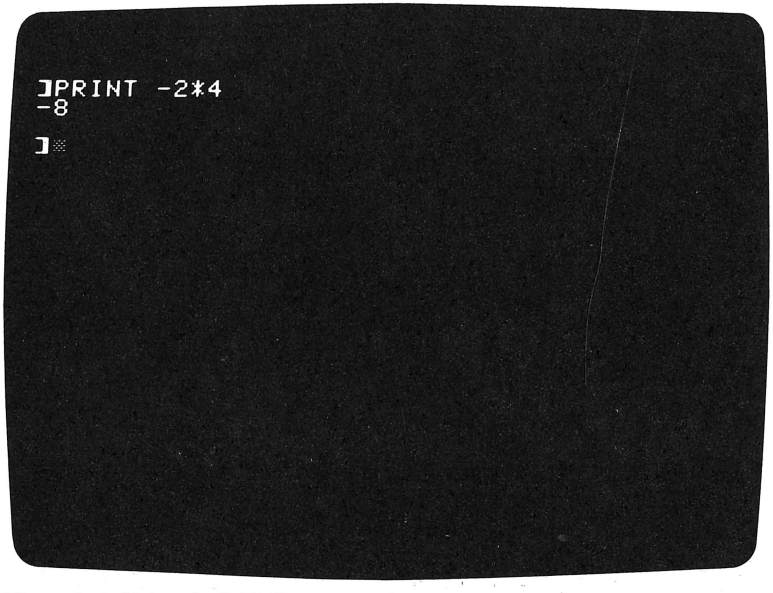


Figure 2.5: Using the PRINT command to see the results of multiplication.

to operate your sophisticated Apple II as a simple calculator, with entries like this:

```
PRINT 9+8-7*6/5
```

where first multiplication, then division, and lastly addition and subtraction are carried out to give the result, 8.6, on screen. Parentheses can be used to algebraically direct the computer's math to work out the values inside them, like this:

```
PRINT (9+8-7)*6/5
```

for an on-screen result that will appear below the PRINT statement as "12."

But the PRINT command's usefulness to you goes beyond displaying mathematical manipulations, the full range of which are listed in Appendix A. PRINT is a command you will find yourself using often and freely. As well as applying it to direct some internal operation onto the screen where you can see it, you can use it to express a sentiment, a thought, or a fact on the screen in the midst of some other computer action. The PRINT command is your typist.

You can direct the computer to print on screen nearly anything you type in quotes by using the PRINT command like this:

```
PRINT "So glad to be here."
```

The computer's response after you press the RETURN key will be the words,

```
So glad to be here.
```

on the line following the command.

As in the rest of the world, words and numbers do mix. For instance, you can make the statement:

```
PRINT "The winning number is..." 30
```

and see the computer reply:

```
The winning number is...30
```

You can go on and on like this, putting spaces inside and next to

the quotes to keep the patchwork readable and even slipping in a calculation, like this:

```
PRINT 20 " or " 30 " is more than I'm willing to pay. But "
10+5 " sounds right. "
```

for the display from the computer:

```
20 or 30 is more than I'm willing to pay. But 15 sounds right.
```

You can even use the PRINT command to make sounds from the Apple II. For instance, the beep you hear each time you switch on or reset the computer can be produced by pressing the CONTROL and G keys together. Treating this CONTROL-G combination as a character itself, you can command the computer to beep by pressing those keys in combination (here shown in parentheses) between the typing of a pair of quotes. CONTROL-G won't show on the screen like other characters, but the computer will follow the command to PRINT it just the same, like this:

```
PRINT "(press CONTROL-G)(press CONTROL-G)(press
CONTROL-G) ATTENTION"
```

to which the computer will respond with three beeps and the display:

```
ATTENTION
```

Once you realize that the computer handles text on the screen as a grid of 40 columns by 24 rows, you can make displays to suit the eye. A handy little command used within a PRINT statement directs the computer to *tab* (or space over to a specified column) before actually printing on the screen. The form is TAB( ), where the columns are numbered, like a typewriter's, from left to right, the rightmost as 40. You can add the command TAB( ) before the item to be printed, like this:

```
PRINT "HEY!" TAB(15) "LOOK" TAB(30) "OVER HERE!"
```

The computer acts on this statement by reading it across as a series of commands, and prints:

```
HEY!          LOOK          OVER HERE!
```

Two punctuation marks can be used as a shorthand command with a PRINT command. Commas typed between items in a

PRINT command will direct the computer to space them three across the screen, as though following a succession of TAB(1), TAB(17), and TAB(33) commands. Semicolons typed between items will separate them in the statement, but not in the printing.

The computer follows all PRINT commands by displaying text or results of calculations as fast as it can, in the same way it replies, which to the eye seems instantaneous. Actually, each letter in a statement is printed on the screen one after another. You can slow down the rate of printing on the screen, whenever you want to draw attention to printing.

Just as there are 40 columns across the screen, there are a number of speeds of printing you can command of the computer, 256 in all. The automatically-chosen fast speed is 255, the slowest is 0. You can switch the computer into printing at either of these speeds or any in between, and any text that follows will appear character-by-character at that speed. You can set the slowest speed with the command:

```
SPEED = 0
```

PRINT directions and "ERROR" replies will then appear one sluggish character after another. You can see the result with a PRINT command like this:

```
PRINT "A SLEEPY LIZARD ON A COLD ROCK IN THE
MORNING"
```

which the computer lays down unhurriedly as an A, then a space, then an S, then an L, and so on.

You can order a more sprightly pace with:

```
SPEED = 100
```

which will lay out such PRINT statements and ERROR replies as the command:

```
PRINT "RUSHING TO MEET A DEADLINE"
```

at the pace of an accomplished typist.

And you can, of course, switch the computer back to instant display with the command:

```
SPEED = 255
```

The SPEED = command works like a switch with 256 possible settings. You can use it with other switching commands like INVERSE or FLASH to tailor the computer's displays of text.

COMMANDS IN GROUPS

Typing one command statement, seeing its effects, then typing another for its effects, and so on, in card-playing fashion, is a method that is fine for controlling text on the screen but is limited in its results and speed. When you want the computer to act on a group of commands, one after another without interruption, you can use a method that sends a series of commands together. The method is a simple one. You type in a command statement, then a colon, then another command statement and another colon, and so on, until you've typed the last command of the series. If you then press the RETURN key, you'll send the whole sequence to the computer at once. Pressing RETURN is your signal to the computer to act on the series in order from first command to last.

You can get an idea of how this multiple statement approach works by typing this sequence of commands:

```
HOME: PRINT "STILL NORMAL" : INVERSE: PRINT "WHAT
HAPPENED?" : FLASH: PRINT "HEY! FLASHING IS HARD
ON THE EYES!" : SPEED = 20: PRINT "EASY NOW, SLOW
DOWN" : NORMAL: PRINT "THAT'S (press CONTROL-G)
(press CONTROL-G) BETTER.": SPEED = 255: PRINT
"BACK TO NORMAL DISPLAY AND SPEED."
```

and then pressing the RETURN key.

The computer will act on this group of commands as follows:

1. First the screen clears.
2. STILL NORMAL is printed instantly at the top-left position.
3. WHAT HAPPENED? is printed below instantly, dark-on-white.
4. HEY! FLASHING IS HARD ON THE EYES! appears next in flashing letters instantly.

5. EASY NOW, SLOW DOWN appears at slow speed in flashing letters.

6. THAT'S appears in white-on-dark letters at slow speed. Two beeps sound slowly. BETTER appears in white-on-dark letters at slow speed.

7. BACK TO NORMAL DISPLAY AND SPEED appears in white-on-dark letters instantly.

You can direct the computer to put the contents of a PRINT statement anywhere on the screen with two location commands: VTAB( ) and HTAB( ). One directs a vertical position at one of the 24 rows; the other directs a horizontal position at one of the 40 columns. These commands, which are similar in form to the TAB( ) command, are distinct from the PRINT statement and direct the computer in placing the flashing cursor. Vertical and horizontal positioners, those commands direct the cursor to a location within the 24-by-40 grid where a PRINT statement can begin displaying.

You can use these position commands like any others in multiple statement groups sent to the computer like this:

```
HOME: VTAB(8): HTAB(15): PRINT "THIS IS THE PLACE."
```

The computer will also act on the VTAB( ) command separately, as a cursor-positioner. You can issue the following set of commands:

```
HOME
VTAB(8)
PRINT "THIS IS THE PLACE."
```

to put the words THIS IS THE PLACE. at the beginning of the eighth row from the top of the screen.

The Apple II is a bit quirky about the way it acts on the HTAB ( ) command: It will do so only if that command is used before a PRINT statement, as in a multiple statement or in a program of statements. If you send that command alone by RETURN key, the computer refuses to carry out its directions, as shown in Figure 2.6. But used in a series of commands, HTAB( ) directs the computer with full force to the column ordered.

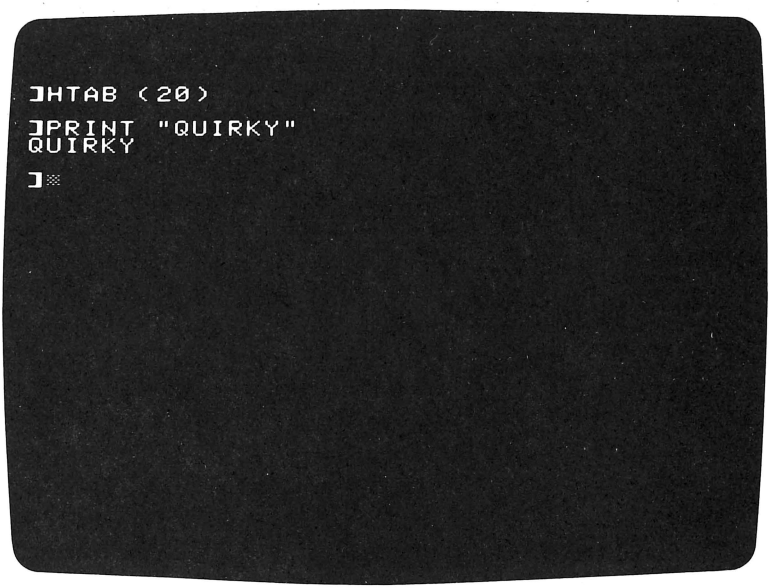


Figure 2.6: The HTAB ( ) command, ignored by the computer when sent as a separate command.

MOSAICS: THE 40-BY-40 GRAPHICS MODE

When you type directly onto the screen from the keyboard and command the computer to PRINT characters onto the 24-row-by-40-column text screen, you're using the computer switched into one of its three basic screen display modes, a text display.

You can switch the computer into another display mode, in which it treats the screen as a grid of 40 columns by 40 rows, into which can be put boxes of differing colors. Following your commands, the computer will draw colors in each of these boxes. These commands, and all other characters you type or send through the computer, will appear in a four-row text window below this chunky, low-resolution grid.

When switched into this graphics mode, the computer blanks the 40-by-40 grid, leaving only the text window at the bottom unchanged. You can switch into 40-by-40 graphics with the command:

```
GR
```

The computer automatically prepares to draw in black, so you won't see any change in the graphics screen until you direct it to use another color. You can switch on a white hue with the command:

```
COLOR = 15
```

in which 15 is the code number the Apple II translates as white. Now, to get some results from the switching you've done with these two commands, you can command a location with column and row numbers. For instance, the computer will draw a box, in this case a white one, at the center of the graphics grid when you send the command:

```
PLOT 20,20
```

You can command white in other box locations by giving a PLOT order with other row and column numbers. In the case of the graphics screen, the computer counts columns and rows from the left and the top, starting with 0. You can plot other points with the following sequence:

```
PLOT 0,0
PLOT 39,39
PLOT 0,39
PLOT 39,0
```

which, incidentally, marks the four corners of the graphics display area.

To switch the computer into drawing with another color, you can use another code number, one between 0 and 15, with the COLOR = command, like this one for purple:

```
COLOR = 3
```

After directing the computer to use another color, you can command that any location be filled with that color:

```
PLOT 19,19: PLOT 21,21
```

As it does with characters on the text screen, the computer will put a new color block over a previous one when you so PLOT it. The command:

```
PLOT 20,20
```

replaces the center white box with a purple one.

If you're looking at a black-and-white television or video monitor, the colors you PLOT will appear as boxes of white dots on a darkened screen. You might notice variations in shading among some of the colors so displayed, and a difference in dot configurations. Within each box, the dots run in a horizontal pattern for white (COLOR = 15) and vertically for other colors.

Since the computer treats black (COLOR = 0) as a color, you can use it to erase colors already on the screen. The commands:

```
COLOR = 0: PLOT 20,20: PLOT 19,19: PLOT 21,21
```

for instance, will erase the boxes at the screen's center.

You can direct the computer to lay down whole strips of boxes at a time with two other commands, VLIN and HLIN. One directs vertical strips, the other horizontal. Between endpoints you designate within a particular row or column, a whole strip of boxes will be drawn at your command. For instance, to lay a vertical line of boxes for the whole length of the leftmost column you can first switch back to white (COLOR = 15) and then command:

```
VLIN 0,39 AT 0
```

This command, read across, can be translated: "Computer, draw a vertical line of boxes beginning at row number 0, and ending at row number 39, in column number 0."

You can fill the uppermost row similarly with the horizontal line command of the same form:

```
HLIN 0,39 AT 0
```

Two more commands for lines will complete the boundary of the graphics screen:

```
VLIN 0,39 AT 39: HLIN 0,39 AT 39
```

for the right-most column and bottom row, respectively.

You might notice that repeating the same endpoint numbers with these commands at different rows and columns will produce symmetric patterns, like that in Figure 2.7, produced by the following commands:

```
VLIN 5,34 AT 10: VLIN 5,34 AT 30: HLIN 5,34 AT 10: HLIN
5,34 AT 30
```

You can switch the computer display back into the text mode

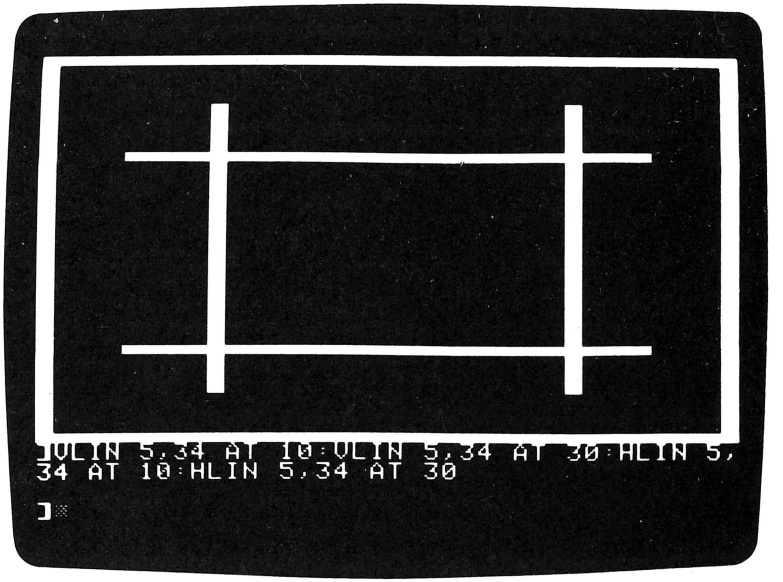


Figure 2.7: Using VLIN and HLIN commands to draw in 40-by-40 graphics.

with another command. The computer will fill the graphics part of the screen with @ characters, then scroll up that odd display from the bottom-most row, when you give it the graphics-to-text command:

```
TEXT
```

Of course, the computer will switch back to the 40-by-40 graphics display on again receiving the GR command. When it switches back, you'll be presented with a fresh, blank graphics screen, all former plotting erased.

SKETCHES: THE 280-BY-160 GRAPHICS MODE

You can switch the computer to another graphics display that's finer than the 40-by-40 mode but offers fewer colors. In this higher-resolution graphics mode, the computer treats the screen as a grid of 280 columns and 160 rows of dots, above four rows of text. The Apple II can print dots in one of six colors, though not every dot can be each color. Through a single command, the computer

can be directed to draw either a single dot, or lines between dot locations.

To switch the computer to this graphics mode, you can use the command:

```
HGR
```

which, like its 40-by-40 counterpart, automatically selects the color black with code number 0.

To switch the color to white, and draw a single dot at screen's center, you can send the commands:

```
HCOLOR = 7
HPLOT 140,80
```

By extending the HPLOT command, a line or series of lines can be drawn from one endpoint to the next, at any angle. At some angles, the lines are straighter than at others. You can draw the boundaries of the 280-by-160 graphics screen with the single command:

```
HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
```

which draws lines from the top-left corner to the bottom-left to the bottom-right, to the top-right, and back to the top-left corner.

You can send the endpoints of diagonal lines that slash across both rows and columns, with commands like this:

```
HPLOT 200,140 TO 260,20
```

By drawing lines among a group of points in different sequences you can create differing displays from the same points.

This set of five points connected by the command:

```
HPLOT 50,10 TO 90,40 TO 70,90 TO 30,90 TO 10,40 TO
50,10
```

will produce the simple five-sided figure of Figure 2.8. After you give the commands

```
HGR: HCOLOR = 7
```

a fresh blank graphics screen is presented. Then you can connect

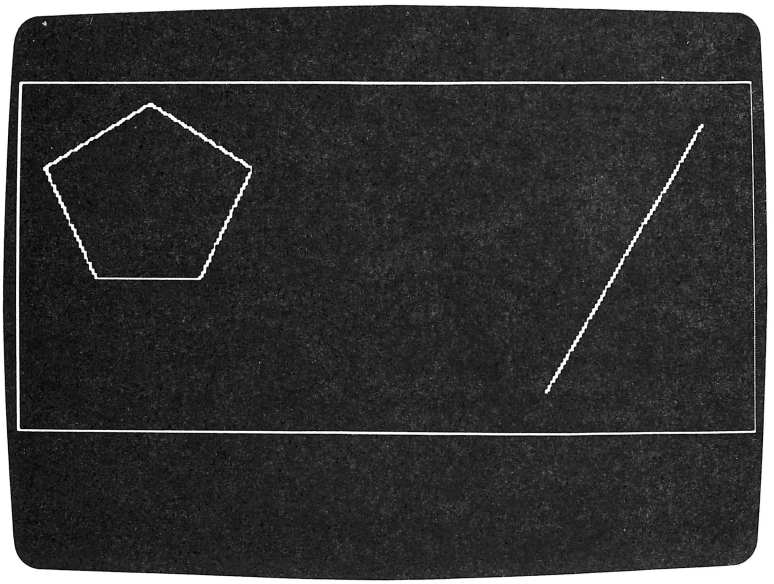


Figure 2.8: Drawings in 280-by-160 graphics, including a simple 5-sided figure.

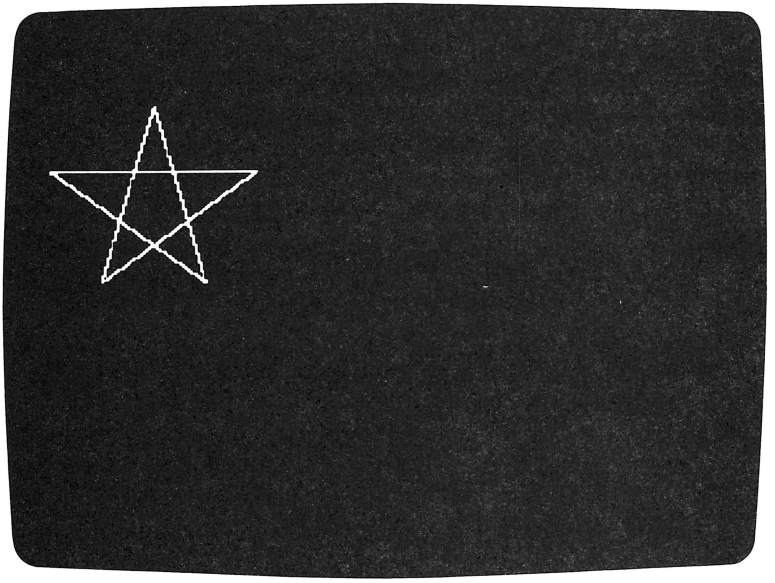


Figure 2.9: The same five points connected in a different sequence.

the same points in a different order with this command:

```
HPLOT 50,10 TO 70,90 TO 10,40 TO 90,40 TO 30,90 TO
50,10
```

to display the star of Figure 2.9.

With a series of commands like the following, you can draw multicolored figures:

```
HCOLOR = 2: HPLOT 20,20 TO 200,20  TO 270,20
HCOLOR = 3: HPLOT 20,20 TO 200,40
HCOLOR = 5: HPLOT 20,20 TO 200,60
HCOLOR = 6: HPLOT 20,20 TO 200,100 TO 270,20
```

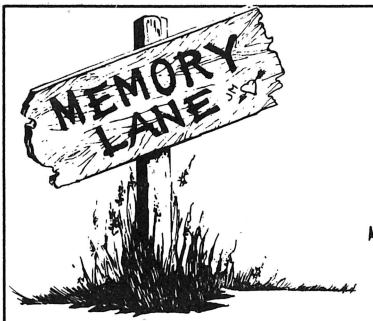
And, of course, you can switch the color to black to erase any plotting from the screen, like this:

```
HCOLOR = 0: HPLOT 20,20 TO 200,100 TO 270,20: HPLOT
200,20 TO 200,160
```

The 280-by-160 graphics display mode is but one of three basic screen displays and can be switched off and replaced by either of the other two with the commands TEXT or GR.

And, you can switch the computer back into 280-by-160 graphics just as quickly with the HGR command again. As it changes from this mode, the computer will lose the display on the screen. If you switch back with the HGR command, you'll encounter a fresh, blank screen. In moving from this display to the text display, you'll find rows of text printed on the upper part of the screen as though they had been pulled up on a scroll above the text window.

By controlling the video screen, you can control the computer's responses to whatever internal trials you put it through.



CHAPTER THREE

SECRET SCHEMES— PROGRAMS

SETTING A SCHEDULE

We mortals, dimly remembering locations, return to them by street addresses. The computer, instantly forgetting the commands you give, returns to them once they're settled in memory addresses.

You can provide such addresses by starting each group of commands with a number. A number placed at the beginning of a group of commands serves as a marker to the computer to retain those commands, possibly to be used again. To mark a statement of display commands this way, you could send the following command, for example:

```
10 HOME: VTAB 12: HTAB 8: PRINT "NUMBERS AND THEIR
   USES"
```

You'll notice, when you press the RETURN key after typing a numbered statement like this, that the computer replies with a right-bracket prompt and flashing cursor, without acting on the commands.

Actually, the statement is sent into storage in the computer's memory, where it is kept on file by number.

In memory and unseen, the commands will stay under number 10 until you type a command that tells the Apple II to proceed with

them. The computer will hold those hidden commands unused until you send the command:

```
RUN
```

which directs it to translate and act on the numbered statements in its memory, in numeric order. As the only numbered line of commands in memory, the statement numbered 10, in this case, will be acted on command-by-command. The computer automatically keeps numbered commands in memory even after acting on them. So you can tell the computer to carry out the commands of line 10 as many times as you like, in this instance, by sending the command RUN each time.

Although commands may reside, unseen, in the computer's memory, you can conjure them onto the screen display with a simple command. If you send the command:

```
LIST
```

the computer will respond by printing the entire contents of its program memory. In this case, that's line number 10.

You can send another set of commands for the computer to act on after line 10 by labeling the statement with a higher number, like this:

```
20 SPEED = 25: FLASH: VTAB 15: HTAB 5: PRINT "THIS LINE
   WILL BE REMOVED."
```

The computer now has two lines in its memory and, when sent the command:

```
RUN
```

will start with the lower-numbered statement, 10, translate and carry out each command, and then translate and carry out the commands of statement 20. You can switch the computer back to the automatic modes of speed and character display after it has completed this RUN with the commands:

```
SPEED = 255: NORMAL
```

Through a series of numbered statements, you can direct the computer to switch among different modes of screen display to

fashion a result otherwise unattainable, like that from these additional statements:

```
30 SPEED = 100: INVERSE: VTAB 20: HTAB 5: PRINT "FOR
   YOUR INFORMATION, YOU ARE NOW PROGRAMMING."
40 SPEED = 255: NORMAL: PRINT " PROGRAM ENDS HERE"
```

If, after sending these statements to the computer, you command RUN, you'll see a display of four statements appearing one after another with a final result like Figure 3.1.

Statement lines 10, 20, 30 and 40 make up a program that can direct the computer, each time you issue the RUN command, to produce the same display printing. Change the statements in the program and you change its results. For instance, the flashing row that declares:

```
THIS LINE WILL BE REMOVED.
```

will be absent from the screen during a program run if you erase program line 20, in which you commanded its printing. To erase a

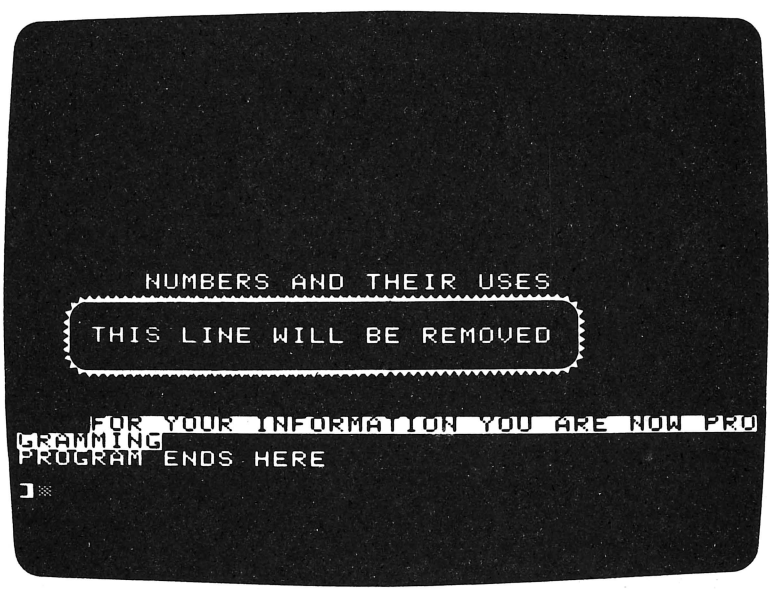


Figure 3.1: The results of running a program of display commands.

line from the computer's program memory, you can simply type the line number and press the RETURN key:

```
20
```

The computer, standing ready to accept a new statement, prepares to store a set of commands under the number 20 when you do this. When you send only the number, it replaces the former statement under this number with the new, blank statement. Since a blank statement gives no direction, line 20 ceases to exist in the program memory.

When you RUN the program, you'll find the flashing text produced by line 20 absent. When you LIST the program, you'll find only lines 10, 30, and 40 remaining in memory, as in Figure 3.2.

The computer follows this simple dictum in handling program lines: The contents of a newly-sent numbered line replace those of the same number previously sent.

You can add a new statement positioned as line 20 like this:

```
20 PRINT "(press CONTROL-G)(press CONTROL-G)(press
   CONTROL-G)"
```

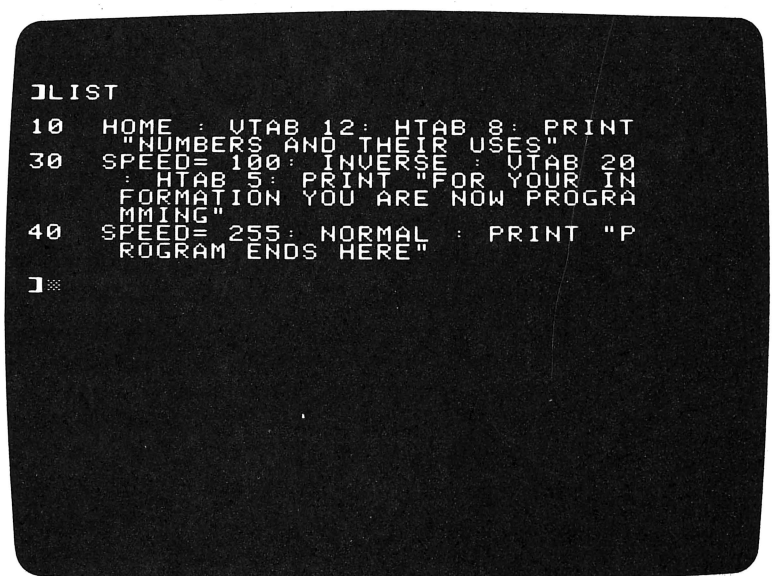

]LIST
10 HOME : VTAB 12: HTAB 8: PRINT "NUMBERS AND THEIR USES"
30 SPEED= 100: INVERSE : VTAB 20 : HTAB 5: PRINT "FOR YOUR INFORMATION YOU ARE NOW PROGRAMMING"
40 SPEED= 255: NORMAL : PRINT "PROGRAM ENDS HERE"
]

Figure 3.2: The listing of the display program, revised.

to include three beeps in the program after the line,

```
NUMBERS AND THEIR USES
```

is printed.

The IIe computer carries out statements sequentially, according to any numbering scheme (of positive, whole numbers up to 63,999), and so will follow a program beginning at any number moving up the sequence of numbers, regardless of gaps in numbering between them. Numbering by tens is a handy approach to programming that leaves room between statements to add program lines that occur to you as afterthoughts or refinements.

The first command in line 30 of this program switched the computer into a slower speed. You can add more sound at this pace with keypress commands by adding a line between 30 and 40, like this one:

```
35 PRINT "(press CONTROL-G)(press CONTROL-G)(press
   CONTROL-G)(press CONTROL-G)(press CONTROL-G)"
```

When the computer runs the program, here's what happens: The screen darkens and the line 10 quote, "NUMBERS AND THEIR USES," appears. The three tones called for in line 20 then sound in rapid succession. The dark-on-white quote from line 30, "FOR YOUR INFORMATION YOU ARE NOW PROGRAMMING," appears at the slow speed. Five tones from line 35 sound at that leisurely pace. And finally, the quote from line 40, PROGRAM ENDS HERE, appears.

The computer retains numbered statement lines in its program memory until you switch off the power (or press the CONTROL/HOLLOW-APPLE/RESET combination), or remove the lines by typing line numbers, or command the computer to erase them all. You can direct this erasure by sending the simple command:

```
NEW
```

On receiving this command, the computer prepares for a new program by erasing all program lines from its memory. If you plan

to write a new program and don't clear out the old program from that memory, old program line numbers not replaced by new ones will tag along from the original program, giving directions you may not have planned for the new program.

You might have noticed that the computer handles program lines limited to the same maximum length as unnumbered statements, 256 characters. And, on the other hand, a program line can contain, like an unnumbered statement, only a single command like HOME. How you group your commands in a program is a matter of personal preference.

If you had written the preceding program one command to a statement line, it would look like this when listed:

```
10 HOME
20 VTAB 10
30 HTAB 10
40 PRINT "NUMBERS AND THEIR USES"
50 PRINT "(press CONTROL-G)(press CONTROL-G)(press
   CONTROL-G)"
60 SPEED = 100
70 INVERSE
80 VTAB 20
90 HTAB 5
100 PRINT "FOR YOUR INFORMATION YOU ARE NOW
    PROGRAMMING"
110 PRINT "(press CONTROL-G)(press CONTROL-G)(press
    CONTROL-G)(press CONTROL-G)(press CONTROL-G)"
120 SPEED = 255
130 NORMAL
140 PRINT "PROGRAM ENDS HERE"
```

Within a program written one command to a statement line like this, you can more easily find problems that call for rewriting or alteration, like the accidental misspelling of a command leading to a "?SYNTAX ERROR" reply during a program run. On the other hand, grouping several commands to a line number may be an easier way for you to think of composing a program from multiple command statements. Either way, the choice is yours.

STAND-INS FOR NUMBERS

Numbers, you've no doubt noticed, are as useful for controlling the computer from within commands as they are in marking program lines. A change in the value of a number changes the action of the Apple II in, for example, the directions given by a VTAB command or the multiplications ordered with an asterisk (*) command.

A single number can be passed around within a program by the computer to be used in different commands and statement lines if you've prepared the computer with the number you'd like acted on. You can do this by using a symbol that stands for the number in the commands which act on it. If you instruct the computer, at a low-numbered statement line early in the program, that a symbol is standing in for a number, the computer will substitute the number at each command in which it subsequently encounters that symbol, and then act on it.

The handiest symbol to use as a stand-in for a number is a letter. You can use the letter in all commands where you'd otherwise have used the number. If you were considering text placement on a screen you might put such a substitute in a VTAB command within a program like this:

```
20 HOME
30 VTAB A: PRINT "HOW DOES THIS LOOK?"
```

The letter A substitutes for a number in the VTAB command here. If you sent the RUN command to the computer for this two-line program without directions as to the value of A, it would follow built-in automatic directions and replace the letter A with the value zero.

By using a simple symbolic command, you can direct the computer to act on a value of your choice. The command takes the form of the equal (=) sign, and to direct the computer to substitute for A a value of 12 in this program you could send the command:

```
10 A=12
```

When you RUN this three-line program through the computer, the result will be the words "HOW DOES THIS LOOK?" printed at row 12.

Having so named a number used in the program, you can change the operation of the program without changing the line that directs that operation. Change the value of A in line 10, like this:

```
10 A=20
```

and the computer will print at row 20. The letter A used this way acts as a *variable* symbol that can stand for whatever number you specify with the equal-sign command.

You can use a variable like this in as many commands in your program as you want. If you add the command:

```
25 HTAB A
```

to the program, the computer will respond by printing beginning at column number 20 and row number 20. Again, if you change the value of A, the computer positions the printing accordingly, by row and column number:

```
10 A=5
```

A variable letter like A can be used any number of times in a program to stand in for a number. In this program you can use it to try out different screen-printing placements by retyping only the single short program line 10. In more complex programs, the effect of changing a variable's value by changing an equal-sign command can be profound.

The assignment command, =, can be translated as a computer direction to work internally to replace the symbol on the left of the equal sign (wherever that symbol is encountered), with the numeric value on the right of that equal sign.

The computer can act on more than a single variable in a program, if you direct it with an assignment statement command for each variable. If you can describe a situation with numbers, you can direct the computer to simulate the situation with each, some or all of its aspects changed. You can, for instance, compare different salary offers or the filling of different dams.

The filling of a dam or the filling of your bank account could be predicted by forming a model of factors known to be at work, and then sending values to the computer with assignment statements to see what the model predicts as a result. If you would like to predict the amount of water in a reservoir or the amount of money in your account after a period of time, nine months for instance, you can

do that simply if you know the amount already there and the average monthly rate at which more accumulates. You can direct the computer to print these values and then to print the calculation of the total amount nine months later with the following program. In it, the original amount is symbolized by the letter A, and the monthly rate of accumulation by the letter B:

```
20 HOME
30 PRINT "STARTING WITH " A
40 PRINT "AND GAINING " B " EACH MONTH"
50 PRINT "YOU'LL HAVE " A+(B*9) " AFTER 9 MONTHS."
```

If you command the computer to RUN this program now, it will automatically substitute zero for the variables A and for B. But if you assign values (through an assignment statement) to match the actual amount present and the rate of accumulation, the computer will display those values and, through the directions in program line 50, print a prediction of the future amount. You can give values to the variables with the program line:

```
10 A=2200: B=355
```

The program will run the complete program, with 2200 as the starting amount (gallons of water, or dollars) and 355 as the monthly increase, to produce the display shown in Figure 3.3.

You can, of course, change the simulation by changing the commands in line 10, and so make predictions for different circumstances. This program, written to produce a prediction for nine months, can be altered by replacing the value 9 with a variable through which you can produce a prediction for any number of months. To do this you can modify line 50, which produces the display and calculation. You can call the number of months the variable C:

```
50 PRINT "YOU'LL HAVE " A+(B*C) " AFTER " C
   " MONTHS."
```

By adding another assignment command, =, you can specify another length of time, say 15 months, like this:

```
10 A=2200: B=355: C=15
```

When the program is run through the computer, it will produce a display and prediction of an original amount of 2200, increasing

by 355 each month, over a period of 15 months.

You can direct the computer to substitute one variable letter for another, or for a set of variables. In this program, if you wanted a shorthand way of naming the final predicted amount you could rename that final amount, so far represented by the value of A + (B*C), like this, calling it F:

```
15 F=A+(B*C)
```

Now that the final value has a simple name, you can command the calculation and printing of the difference between the original and final values, A and F, with this simple statement:

```
60 PRINT "THE GAIN IS " F-A
```

You could then streamline the program's appearance a bit with a substitution in line 50:

```
50 PRINT "YOU'LL HAVE " F " AFTER " C " MONTHS."
```

You can still change the details of this model by changing the values in line 10.

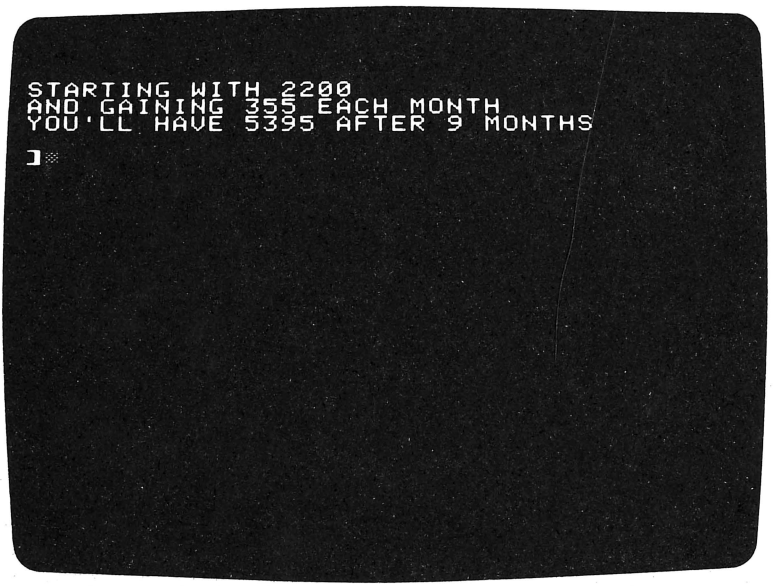


Figure 3.3: Display of the results of running a computer simulation.

By the way, the computer is designed to work with decimal numbers and so can take values given in fine detail if you wish, like A = 2200.35, and can handle fractions if they're treated like divisions, say for 10 and 3/4 months, C = 10 + 3/4.

MAKING YOUR COMPUTER DECISIVE

You can watch over your Apple II, deciding when you'd like certain commands repeated, or variable values changed, or when you'd like to interrupt a program run. That's effective, all right, but means that you're attending to the computer rather than the other way around.

A useful command would reflect your decisions inside a program so that when the computer encountered circumstances you'd like acted on, it would take action. Directions to follow such a command are built into the Apple II, which will examine a numeric comparison, like A>5or B = 89, and take the action you direct. If such a condition is not met, the Apple II will skip over the rest of this command and act on the next program line in sequence.

This conditional command follows the logic of everyday English, as in the conditional statement, "If it's raining now, then go by car." In fact, the computer command takes the same "if, then" form. As such it serves two functions, one after the other. First, as in English, the circumstances expressed after the "if" part of the statement are evaluated. When they're true, the action called for in the "then" part of the statement is taken. When not, that action is skipped over.

You can use the IF-THEN command in a new program to evaluate a sum of three numbers represented like this:

```
10 A=5: B=8: C=87
20 T=A+B+C
```

You can arrange for the computer to respond with a display if the sum is 100 with the command:

```
30 IF T=100 THEN PRINT "A SUM OF A HUNDRED"
```

which will direct the printing of the quote only when the condition T = 100 occurs. (Notice that the equal sign is *not* being used as a command here.)

When this program is run through the Apple II, the quote will

be printed. If line 10 is changed so that the values add to something else, the computer will skip past the THEN part, in this case to the program's end.

In encountering an IF-THEN statement, the computer follows built-in directions to evaluate the condition stated to the right of the word IF. When that condition is true, the computer follows the command to the right of the word THEN. When it is false, the computer skips to the next program line.

You can direct the computer in this decision-making process through any mathematical relation it is built to understand. This change to the program, for instance, tests the numbers of guests arriving in groups at a gathering.

```
30 IF T > 100 THEN PRINT "THE ROOM IS GOING TO BE
   CROWDED"
```

Facing an IF-THEN command, the computer is put at a crossroads where it automatically acts on the command there, or skips to the next line. When encountering a number after the word THEN, the computer translates the command as a direction to carry on with the program from the commands in that line number.

Besides instantly following a command in the IF-THEN statement, the computer can be sent to another part of the program by line number, as in this program:

```
10 A = 20: B = 15: C = 9
20 T = A + B + C
30 HOME: PRINT "WITH GROUPS OF " A " AND " B " AND "
   C "THE ROOM WILL HAVE " T " PEOPLE IN IT"
40 IF T > 75 THEN 100
50 VTAB 12: PRINT "NO PROBLEM ACCOMMODATING THAT
   MANY (press CONTROL-G)(press CONTROL-G)"
60 END
100 VTAB 18: PRINT "THAT'S MORE THAN THE ROOM WILL
   HOLD! (press CONTROL-G)(press CONTROL-G)(press
   CONTROL-G)"
```

Using the values given in line 10, the computer will print out the quotes in lines 30 and 50, then stop carrying out commands on reaching the END statement, skipping past the command which would redirect it to line 100 from line 40, THEN 100. If you

change line 10 so that the sum exceeds 75:

```
10 A = 19: B = 23: C = 37
```

and run the program, the computer will carry out the command after THEN and skip past lines 50 and 60 to carry out the print statement of line 100.

The Apple II will also respond to directions to line numbers from an IF-THEN statement the same way when the statement is written in the more descriptive form, IF-THEN-GOTO, as in this rewritten line 40:

```
40 IF T > 75 THEN GOTO 100
```

which produces the same results as its predecessor.

Directing the computer forward through a program according to values it encounters or directing it to some immediate command within an IF-THEN statement, you can change the outcome of computer action depending on the values encountered in an IF-THEN statement. By directing the computer backward to an earlier line number from an IF-THEN statement, you can make it repeat earlier commands, with variations if you like.

This backward-pointing finger from an IF-THEN statement you can apply, for instance, to the problem of filling a party room with guests arriving in batches without exceeding the room's capacity. In a situation like this, there are three values to consider: the capacity of the room, which you can name as variable C; the number of guests at the party, which you can call G; and the rate at which new guests arrive each trip—call this R. If you'd like to know how many trips will fill but not crowd the party, you can use an IF-THEN statement to direct the computer into considering trip after trip, and then signalling you when the simulated trip that fills the room occurs.

You can begin this program with a statement defining the room size, the number of guests at the simulation's beginning, and the rate of new arrivals:

```
10 C = 200: G = 20: R = 6
```

The computer can prepare the screen and display the available room from these commands:

```
20 HOME: VTAB 10
```

```
30 PRINT "AFTER " N " TRIPS " G " PRESENT & ROOM FOR "
   C-G
```

In line 30 the variable letter N will serve as a counter for the number of trips through which the program has directed the computer. You can make N a counter and keep G, the number of guests, up to date with these commands:

```
40 N = N + 1: G = G + R
```

And then the crucial command directs the computer to continue or to stop considering trips:

```
50 IF G < C THEN 30
```

This directs the computer to either return to the sequence at line 30 (if G is less than C) or continue to the final line, which signals you that a limit has been reached:

```
60 PRINT:PRINT "(press CONTROL-G)
   (press CONTROL-G)(press CONTROL-G) - - - - - FILLED
   AFTER TRIP # " N
```

When run, this program will direct the computer to repeat the quotation in line 30 with different values until the room's limit is reached. Here's what happens through these lines, as shown in Figure 3.4.

The computer puts the values for line 10 into its memory for C, G, and R, then clears the screen and positions the cursor for the line 30 PRINT statement. When that statement is first encountered, the value of N has been automatically set at zero. This represents the condition of the room before the first trip of newcomers:

```
AFTER 0 TRIPS 20 PRESENT & ROOM FOR 180
```

Encountering line 40, the computer considers trip number 1, which increases the number of guests to 26. Then, at line 50, it compares the number of guests to the room's capacity and finding on this first pass that the number is smaller, it follows the IF-THEN direction to line 30. Acting again on line 30, the computer displays the values of N and G after that first trip:

```
AFTER 1 TRIPS 26 PRESENT & ROOM FOR 174
```

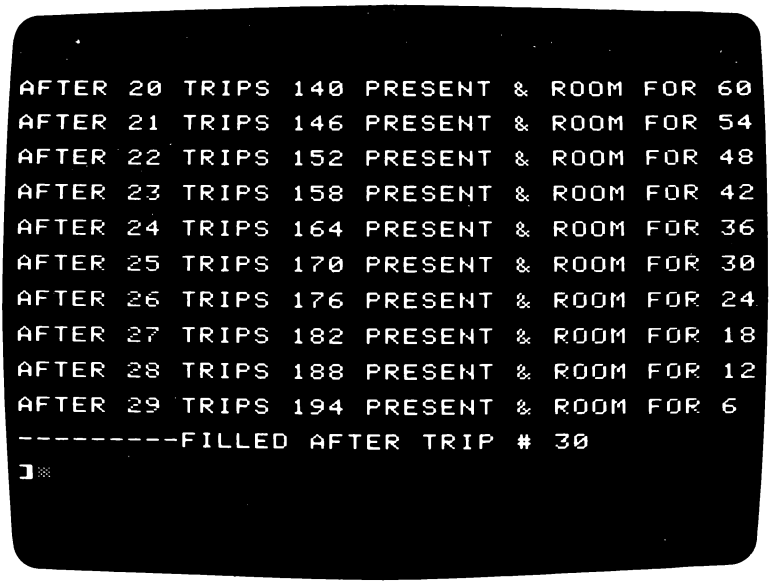

AFTER 20 TRIPS 140 PRESENT & ROOM FOR 60
AFTER 21 TRIPS 146 PRESENT & ROOM FOR 54
AFTER 22 TRIPS 152 PRESENT & ROOM FOR 48
AFTER 23 TRIPS 158 PRESENT & ROOM FOR 42
AFTER 24 TRIPS 164 PRESENT & ROOM FOR 36
AFTER 25 TRIPS 170 PRESENT & ROOM FOR 30
AFTER 26 TRIPS 176 PRESENT & ROOM FOR 24
AFTER 27 TRIPS 182 PRESENT & ROOM FOR 18
AFTER 28 TRIPS 188 PRESENT & ROOM FOR 12
AFTER 29 TRIPS 194 PRESENT & ROOM FOR 6
---------FILLED AFTER TRIP # 30
]

Figure 3.4: Simulation of guests arriving at a gathering.

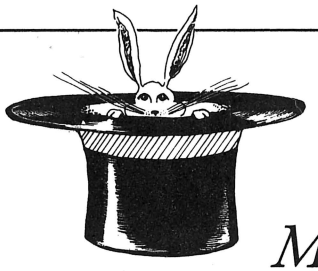
Onward works the Apple II, blind to its poor grammar, considering the next trip, number 2, at line 40 and repeating the process. On and on it works, printing the trip number and condition of the room at each trip until the pass at which the number of guests reaches the capacity, when N is 30. On encountering the IF-THEN command at that point, the computer advances to line 60, sounding three tones as a signal and printing the message that the room would be filled after that trip.

You can add a handy feature to this program by directing it to show the conditions you gave the computer in line 10:

```
70 LIST 10
```

which will print line 10. You can now retype this line to reflect a larger room, C, a different number of guests already present, G, or a different rate of arrival, R.

Under your direction by an IF-THEN command sending it backward, the computer conjures its repetitive powers. Acting under yet other commands, the computer can be sent in any direction to other commands from one address to another. Complex routes can be formed between program-line addresses, each of which asks the computer to do something different.



CHAPTER FOUR

ALL MANNER OF MANIPULATIONS

A PROGRAMMER'S KIT OF COMMANDS

We're only human: We change our minds, we make mistakes, we are prone to forget. The programs we devise and use are subject to our own interesting and imperfect nature.

To allow for this nature, you can mark, stop, open, and cut out whole sections of the programs you use. You can use programming devices from other programs that are useful, but not quite what you're after. In these cases it's helpful to know what the program is doing, in order to tinker it into the shape you want.

One command often handy if you put together many programs will mark program lines without affecting the computer's operation. That command is REM. You can appreciate its usefulness by considering the following graphics program, which directs the drawing of a 40-by-40 checkered pattern, like that shown in Figure 4.1. The pattern is made by first drawing vertical lines in white in even-numbered columns, in program lines 20 through 50—then slicing through those vertical lines with horizontal lines of black in program lines 60 through 90.

```
10 GR
20 COLOR = 15
30 VLIN 0,39 AT X
40 X = X + 2
50 IF X< 40 THEN 30
```

```
60 COLOR=0
70 HLIN 0,39 AT Y
80 Y=Y+2
90 IF Y<40 THEN 70
```

Instead of running the program through the computer each time you wish to see what it does, you can identify it with a labeling program line, which is a note to whoever reads the program, but is not acted on by the computer. To so mark this program, you can add a line that starts with the inaction command REM:

```
5 REM A PROGRAM FOR THE APPLE TWO THAT MAKES A
  40 BY 40 CHECKERED PATTERN.
```

When the computer encounters this REM line in a program it takes the command to mean: "Move on to the next program line; what follows is a remark for the person reading this program." Although a program may make perfect sense to you when you type it, a few weeks later—or sometimes a shocking few minutes later—it

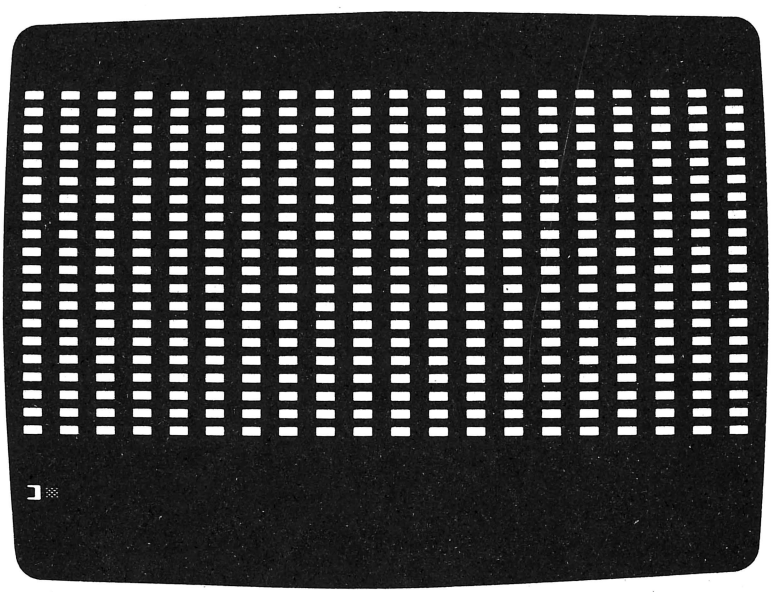


Figure 4.1: A checkered pattern drawn in 40-by-40 graphics.

may seem nothing but a perplexing assortment of commands unless you've tagged it with a note.

The REM command can be put anywhere a program line will fit, so you can describe sections or individual command lines of a program if you like. For instance, in the case of the checkered program, you can add markers before those lines that direct the vertical lines and before those lines that direct the horizontal.

```
15 REM THE NEXT FOUR PROGRAM STATEMENTS DRAW
   20 VERTICAL LINES IN WHITE.
55 REM THE NEXT FOUR PROGRAM STATEMENTS DRAW
   20 HORIZONTAL LINES IN BLACK.
```

A program marked like this will be easier to identify and alter.

You can also follow a program's progress actively as it is run, by switching the computer with a command that directs it to display the line number of each program statement on the screen as the computer acts on that statement. The computer stays in this "tracing" mode until switched out of it. You can switch the mode on by sending the command:

```
TRACE
```

If you run the checkered program after giving this command, the line numbers of the statements will be printed through the text window as the computer draws. You can follow the program loops of lines 30 through 50 as the computer acts on those commands, and of lines 70 through 90. Since this happens very quickly, you might want to slow the action. The graphics-mode commands are not controlled directly through the SPEED = command. But since the computer, once switched into TRACE, is printing, you can control the speed of the graphing by controlling the speed of the printing of line numbers, with the command:

```
SPEED = 5
```

You can watch the computer, so switched, graphing on the screen one command at a time. At this point you've switched the computer into three modes at once: graphics, tracing, and slow speed.

To switch it out of the tracing mode you can command:

```
NOTRACE
```

and the computer will run the program as before, without line numbers and at a fast clip.

Incidentally, you can use the TRACE and NOTRACE commands within a program, like this:

```
15 TRACE
```

to turn on the tracing of the graphing in this program, and:

```
95 NOTRACE
```

to turn it off.

Tracing, often useful when you wish to examine a program, can be a first step to changing computer performance. You may wish to look at an operation only for some interval of program line commands such as the drawing of black lines in program statements 70 through 90. The choice is yours, to command the computer into the TRACE mode for the whole program run or to TRACE more selectively, by bracketing the lines in question with the on and off commands.

If you switch the computer back to full speed ahead:

```
SPEED = 255
```

and clear the two TRACE lines:

```
15
95
```

you can then look at only lines 70 through 90 in more detail by adding two pairs of switching commands:

```
65 TRACE: SPEED = 5
95 NOTRACE: SPEED = 255
```

and then running the program.

While it's running, you might decide you'd like a slightly different pattern. To stop the program in mid-stride you can can use one of three commands. The planned way is to include a command to that effect within the program itself. The command that will do just that is STOP. You can strategically include it to stop the program run after the last of the vertical lines has been drawn by program statements 30 through 50, like this:

```
55 STOP
```

If you run the program with this additional line, the computer, on encountering the statement, will stop following program commands and will inform you that a forced break in the run has been ordered, with the reply:

```
BREAK IN 55
```

This STOP command differs from the END command only in that the computer replies. STOP could be translated : "Print BREAK IN and this line number, and wait for directions from the keyboard."

The really useful advantage to the STOP command is that you can, at your convenience, command the computer to proceed with the program, by sending the command:

```
CONT
```

which could be translated: "Resume carrying out commands at the program line following the interruption."

The second way to stop a program that is running is with the keystroke command that mimics the STOP command in an immediate, unplanned way. To stop a program at any point in its operation, press together the CONTROL and C keys. After receiving this keyboard command, the computer will reply with the "BREAK IN" response. The computer, once stopped in a program run with this CONTROL-C key command, will also resume carrying out program commands when sent the command:

```
CONT
```

By the way, the computer automatically remains in whatever mode it was last switched into. If, for instance, you stop the run after line 65 (TRACE: SPEED = 5), but before line 95 (NO-TRACE: SPEED = 255), and then start it again with the CONT command, the computer will continue tracing and running slowly until it's instructed otherwise.

The STOP command, implanted in a program, and the CONTROL-C command, sent from the keyboard, both affect the computer's operation in the same way.

You can use either one to stop the computer at program line 50, just after the last vertical line has been drawn, to consider varying the pattern. One way to change the display is to select another

color for the rows sketched by the commands of program lines 60 through 90. Even a black and white screen will show changes in shadings among colors.

Interrupting a program can sometimes give you ideas. For example, you can change program line 90 so as to draw only horizontal black lines as far down as row number 25:

```
90 IF Y< 26 THEN 70
```

You can change to horizontal white lines like this:

```
60 COLOR = 15
```

Of course, you can go on and on with revisions until the pattern is one you like, perhaps this one, in which program lines 60, 70, 80, and 90 have been altered.

```
60 COLOR = 9
70 HLIN 5,35 AT Y
80 Y = Y + 3
90 IF Y< 35 THEN 70
```

All these changes produce a centered, thatched pattern. One more command will start the horizontal drawing down a bit from the top instead of from the initial Y = 0, the assumed first value chosen automatically by the computer. You can tell it instead:

```
65 Y = 5
```

The result is a cross-hatched pattern contained within rows 5 through 35, as in Figure 4.2.

A third way of stopping a running program, the most intrusive, directs the computer to leave the program completely, retaining it in memory, but dropping its marker in the run. You can stop a program run this way with an emergency-brake command that operates from the keyboard. To use it, you press the CONTROL and RESET keys together. Since the computer forgets its place in the program, the CONT command isn't a useful way to pick up the sequence again in this case.

A final tool, a command useful in tinkering with programs, will eliminate whole sections of programs. With this command you can ruthlessly expunge as many program lines as you like. For instance, if you wanted to eliminate the vertical lines in the pattern drawn with this program, you could remove program lines 20

through 50, which direct the computer in drawing those vertical lines. Here's a way to do that with a single command:

```
DEL 20,50
```

Run the program after this command and you'll find that only the horizontal lines remain. As used above, the DEL command has the same effect as sending blank numbered lines to the computer, like this:

```
20
30
40
50
```

The DEL command, translated, reads: "Delete from program memory all statement lines starting with the first number and ending with the second."

These commands complete the full set of devices you now have for restructuring existing BASIC-language programs. There are more commands for building the programs themselves.

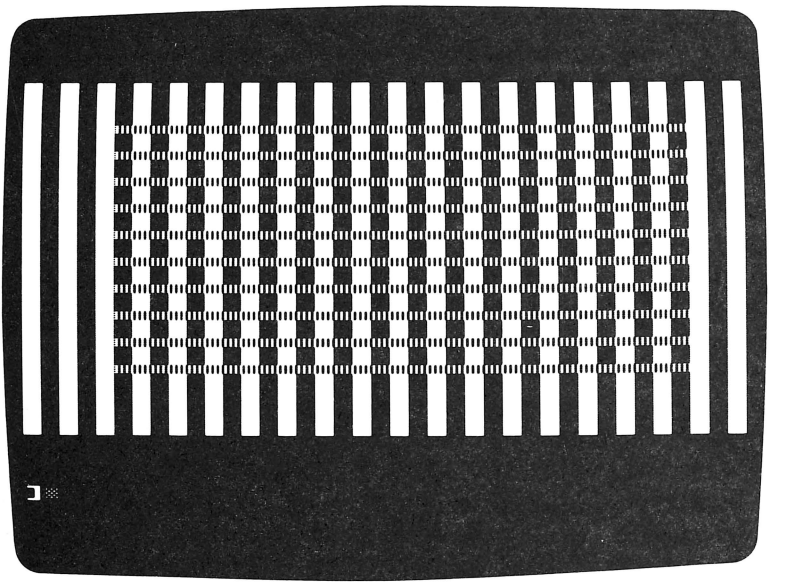


Figure 4.2: A thatched pattern produced by changes to the checkered pattern in 40-by-40 graphics.

COMMANDS FOR MOVING AMONG STATEMENTS

Yet more commands direct the Apple II outside automatic advancement through higher and higher line numbers in a program. One, GOTO, is even simpler in action than the IF-THEN command, and is contained within a version of that command, IF-THEN-GOTO. You can use the GOTO command by itself, to direct the computer back to some earlier statement in a program, again and again, creating an unending *loop* of computer action. Ideal for repetitions, this sort of loop can be stopped only with one of the three program-stopping methods described in the previous section.

The following program will start the computer producing an endless multiplication and division table. In line 10 the number to be multiplied and divided is set at five.

```
10 A=5
20 SPEED=100
30 HOME: PRINT" - - - - - -THE NUMBER " A
   " - - - - - - " : PRINT
40 PRINT " N " TAB(5) "MULTIPLIED BY N" TAB(22)
   "DIVIDED BY N" :
50 PRINT
60 N=N+1
70 PRINT N TAB(10) A*N TAB(25) A/N
80 GOTO 50
```

The final command in program line 80 directs the repetition of the program from line 50 on. Without that GOTO direction, the program would run a short three rows of printing—the heading from line 30, the subheading from line 40 and the first pass at the counter number N (when it's raised from zero to one), the multiplication of A*N and division of A/N from line 70. However, when the computer finishes with the commands of line 70 and translates the command GOTO 50 on line 80, it jumps into the sequence at line 50, thereafter automatically following increasing line-number statements until again encountering line 80, which directs it out of sequence to line 50 again. Each time the computer acts on the counter line, 60 N = N + 1, a new value is applied to the printing, multiplication and division of line 70.

The commands between program lines 50 and 80 will consequently be acted on again and again by an inexhaustible computer until you stop it with a CONTROL-C, CONTROL-RESET command from the keyboard or, of course, by switching off power.

Although you could also direct the computer to skip forward through the program number sequence to a higher-numbered statement line, there's no advantage in doing so, since the intervening statements would be missed forever by the computer. Although a second GOTO command could send the computer back to an earlier line that had been skipped, this type of programming offers no control that couldn't be achieved more simply by a single GOTO command controlling the type of repeating loop seen above.

In this program you can give any value to A in line 10, and you can observe the values as they appear on the screen if you're looking for something particular. Or if you'd like to be signalled, for instance, when the division drops below some value, two variations of the GOTO command can direct the computer to do that looking for you. Both of these commands are fairly rigidly structured, but easy to use once you get the gist of their format. Through one, the ON-GOTO command, the computer is directed to compare variable symbols with a built-in number range, then is sent to a program line. It works much like an IF-THEN-GOTO statement, except that the condition in the first half of the command is preset to ranges of numerical value: 1 or greater but less than 2, 2 or greater but less than 3, and so on. The directions in the GOTO half of the command point to line numbers you arrange according to those preset ranges.

If you wanted the computer to signal you when the value of the division A/N, for instance, falls below two, three, and four you could add a command to do this after line 70 like this:

```
75 ON A/N GOTO 100,200,300
```

which directs the computer to evaluate A/N, compare it with built-in ranges of one or greater but less than two, two or greater but less than three, and so on, and then to proceed with the program line listed, 100, 200, or 300, according to that value. In this case, starting at program lines 100, 200, and 300 you can put signals to alert you to the values produced in by earlier lines of the program. If you had no interest in values less than two, you could give the

command to signal and stop the program, like this:

```
100 PRINT TAB(20) "(press CONTROL-G)(press CONTROL-G)
    (press CONTROL-G) VALUE BELOW TWO "
110 STOP
```

which will sound three beeps, print "VALUE BELOW TWO" in the A/N column, and then stop the computer with the message "BREAK IN 110."

Likewise, if you wanted the computer to continue the program run, and merely to signal you when the value A/N slipped below four and then again below three, you could add statements starting at 300 and 200, respectively:

```
300 PRINT TAB(20) "(press CONTROL-G)(press CONTROL-G)
    VALUE BELOW FOUR"
310 GOTO 50
```

and

```
200 PRINT TAB(20) "(press CONTROL-G)(press CONTROL-G)
    VALUE BELOW THREE"
210 GOTO 50
```

Each pair of statements here signals the dropping value you're looking for, then directs the computer back into the multiplication-division-printing loop at line 50.

You can use the ON-GOTO command to search out and act on values. By putting variables into a form that falls within the rather particular ranges of the first half of this command, you can direct the computer into diversions and exits from normal sequential program runs.

You can specify action on any number of ranges. Line-number directions, each matched to a particular range, can be used to send the computer to work on many more statements. The ON-GOTO command follows this plan: each line number listed after the word GOTO corresponds in step to progressively rising ranges of the variable after the word ON.

You could use a statement like:

```
ON A/N GOTO 100, 200, 300, 400, 500, 68, 97, 12
```

and so on, to cover larger and larger values of A/N.

Numbers, like mules, are not always agreeable, especially when

they lose touch with your intentions. The Apple II, like any well-designed computer, was built to recognize departures from reality that numbers sometimes take, and to reply with an error message when the numbers don't make sense. An old mathematical bugaboo, and one that might creep unseen into an occasional program you use, is the problem of division by zero.

Just about all divisions make numeric sense, even those by very small numbers like one millionth. (How many microbes a millionth of an inch long will fit in five inches? Five million.) But to divide by zero is a puzzler. Some persons interpret it as infinity, which is about as useful as title to a unicorn.

When the Apple II encounters an error, such as division by zero, it automatically stops the program and sends an error message to you on the screen. When you don't want a program stopped by an error, you can add a pair of commands that directs the computer to a separate set of commands and then to return to the sequence.

If you wanted the multiplication-division table to begin at a negative number, -10 for instance, and advance past there, you could start the value of N at -10 with the statement:

```
15 N = -10
```

When this program is run through the computer, a division by zero will occur as the number increases through the N = N + 1 statement of line 60. Unless you've given directions otherwise, the computer, on confronting this division by zero, will stop the run. You can keep it running by using an error-sensing command, ONERR, to direct the computer, as follows:

```
75 ONERR GOTO 400
```

Then, if you add other directions at line 400, you can call attention to the division by zero and keep the program running:

```
400 PRINT TAB(25) "INFINITY OR WHAT ?" : N = 1
410 RESUME
```

When directed by an ONERR-GOTO command, as in line 75, the computer marks the line number of its diversion and, after carrying out statement-line commands through lines 400 and 410, will be sent, by the command RESUME, to pick up the sequence at the point of departure.

The ONERR-GOTO, RESUME command pair can aid in taking out numerical aberrations or other unmanageable surprises in a program. The computer will follow the ONERR-GOTO direction to any program line. You might put it at the beginning of a program when you're trying to alter the error responses. It responds to any error.

REPEATING WITHIN LIMITS

You may find a useful set of commands you want repeated, perhaps not endlessly (as done by the GOTO statement) nor conditionally (as done by the IF-THEN statement), but simply for a specific number of times. The Apple II will respond to a pair of commands, one counting off the repetitions, the other directing the computer back to the first command. You can slip these commands in, one before a stretch of statements you'd like repeated, the other after.

Here's a program to which you can add this command pair in a variety of ways to produce a variety of displays. It directs the computer to draw an arrow shape in 280-by-160 graphics.

```
10 HGR
20 HCOLOR=7
30 X=10: Y=10
40 HPLOT X+15,Y+30 TO X+50,Y+30 TO X+40,Y+20 TO
   X+40,Y+40 TO X+50,Y+30
```

Through the values you type for X and Y, as column and row numbers in line 30, you can locate the arrow drawn by the directions in line 40 anywhere on the screen. Now, to draw the arrow several times at different places, the computer needs directions to repeat line 40 for different values of X and Y, so that the same pattern is plotted from differing starting points.

You can order this with the first statement of the counting pair. In this statement you state a starting value, an ending value, and the size of the jumps the computer should make in going from one value to the other. In the case of the graphics screen, you can specify a column number to be changed at each pass, so that the arrow will be drawn at each of several places along a single row, spaced

50 columns apart, with the statement:

```
35 FOR X = 0 TO 210 STEP 50
```

The other member of this command pair will automatically direct the computer back to the program line on which the FOR-TO-STEP command is found:

```
45 NEXT
```

When you run this program, all program lines numbered between the FOR-TO-STEP and NEXT statements at lines 35 and 45 will be repeated until the limit set after the word TO is reached, as in Figure 4.3.

In this case line 40, the drawing command, is repeated. The computer will begin drawing at each of five locations along a single row (Y = 10). The first starts at the position set by row, column values of 0,10 and the last at one set by 200,10. The value of X assigned in the FOR-TO-STEP statement replaces the earlier one from line 30.

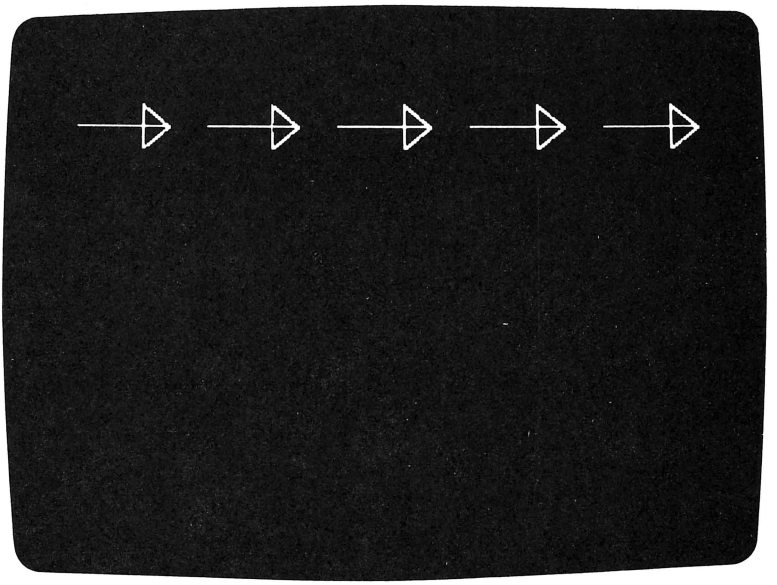


Figure 4.3: An arrow drawn repeatedly in 280-by-160 graphics with a pair of FOR-NEXT commands.

You can give yourself clues to see which commands the computer is following, by switching it with the command combination:

```
TRACE: SPEED = 100
```

then by judiciously using the CONTROL-C stop command and CONT, to examine the program run wherever you like. Remember, you can switch back to the usual automatic mode with the command combination:

```
NOTRACE: SPEED = 255
```

Now, using the FOR-TO-STEP, NEXT command pair, you have a program that directs the drawing of a row of arrows. With another pair of FOR-TO-STEP, NEXT commands ranging over values of Y, you can expand that program to draw a similar group of arrows at more rows to fill the screen with them. To repeat the first row of arrows, that second pair should include the commands that draw it, the FOR-TO-STEP and NEXT pair for the values of X. Thus, the arrows at each X location will be drawn for each Y location. These statements, which range over rows in jumps of 40, are:

```
32 FOR Y = 0 TO 120 STEP 40
47 NEXT
```

The program now in memory directs the computer to repeat the commands that produced the first row of arrows, to measure from the first row (Y = 0) and draw a row of arrows, then move down another forty screen rows to measure from the second row of arrows (Y = 40) and draw another row of arrows, and so on until the screen row of 120 has been reached. Then the computer stops drawing, and four rows of five arrows each fill the screen, as shown in Figure 4.4.

Since each FOR-TO-STEP command sets the initial value of X and Y, the effect of line 30 is canceled, and that line can be dropped:

```
30
```

If you switch to TEXT and LIST the program now you'll see the nested pairs of FOR-TO-STEP and NEXT commands as they appear in Figure 4.5.

Actually, the first NEXT statement, at line 45, directs the computer to take action on line 35, where the value of X is changed. As it is, the computer works without problems, distinguishing one NEXT statement's directions from the other's. You can keep track of the FOR-TO-STEP, NEXT groupings more easily, by labeling the NEXT statement lines to match their FOR-TO-STEP lines, if you like, this way:

```
45 NEXT X
47 NEXT Y
```

You can envision the X FOR-TO-STEP, NEXT loop as "nested" inside the Y FOR-TO-STEP, NEXT loop. When one loop is nested within another in a program, the commands in the inner loop are carried out first, as many times as detailed in the FOR-TO-STEP command, at the first value of the outer loop, then through the second of the outer loop, and so on.

Here's a summary of how the computer is directed by this program:

In line 10 and 20 the higher-resolution graphics mode is switched

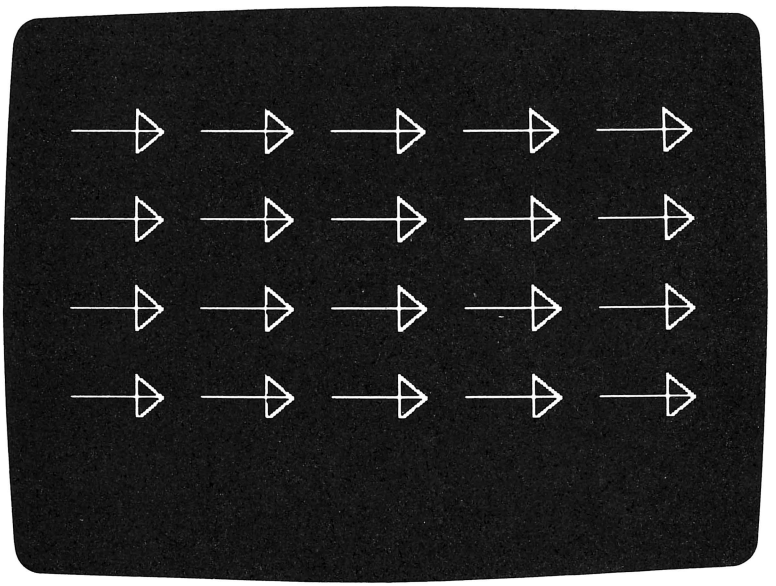


Figure 4.4: An arrow drawn repeatedly in 280-by-160 graphics with two pairs of FOR-NEXT commands.

on and the electronic "paintbrush" is dipped in white. In line 32 the computer prepares to substitute the value zero for each Y it encounters. In line 35 it prepares to substitute zero for each X. In line 40 the substitutions are made, and the resulting points are connected by drawn lines. That produces the first arrow on the screen.

From line 45, the computer is sent back to the FOR-TO-STEP statement of line 35 where the value 50 is added to the last X, and the new value of X is substituted in line 40, so that the second arrow is drawn as measured from the point of column number 50 and row number 0.

When it next encounters line 45, the computer is sent again to the FOR-TO-STEP statement in line 35, and the process repeats with X = 100 and Y = 0 substituted in the HPLOT command.

This goes on until the value of 210 for X is passed. That happens after the fifth arrow is drawn, when the NEXT X statement of line 45 passes the computer on to the following line number 47. Line 47 sends the computer to the FOR-TO-STEP statement at line 32, where 40 is added to the value of Y. With Y = 40, starting

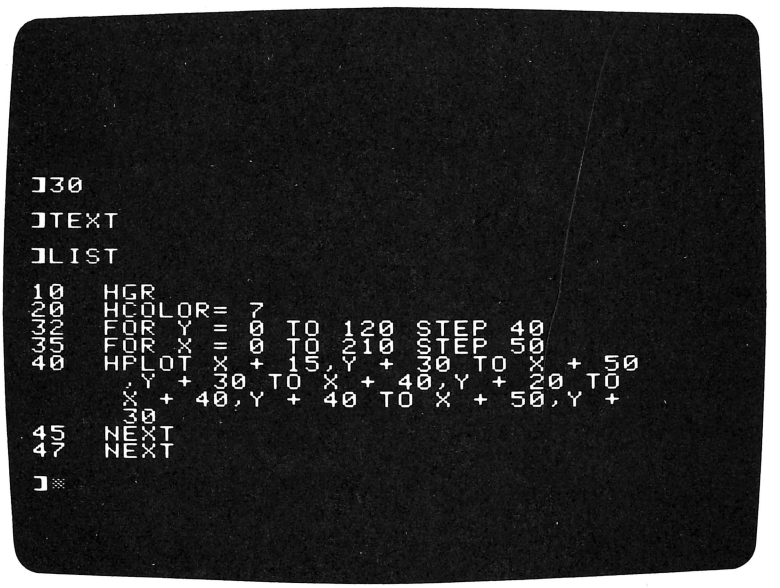

]30
]TEXT
]LIST
10 HGR
20 HCOLOR= 7
32 FOR Y = 0 TO 120 STEP 40
35 FOR X = 0 TO 210 STEP 50
40 HPLOT X + 15,Y + 30 TO X + 50
,Y + 30 TO X + 40,Y + 20 TO
X + 40,Y + 40 TO X + 50,Y +
30
45 NEXT
47 NEXT
]

Figure 4.5: The program of nested FOR-NEXT commands that produce the arrow display of Figure 4.4.

the second row of arrows at screen row number 40, the computer goes through the FOR-TO-STEP statement of line 35 again and emerges, dropping through line 45 when the second row has been completely drawn.

The FOR-TO-STEP, NEXT Y statements of line 32 and 47 then keep the computer busy on the third, and then the fourth, rows of arrows until its values are exhausted, after the fourth row, when the value of 120 is passed. Thus, you are left with a screen full of four rows of five arrows each.

As you might suspect, if two nested loops of FOR-TO-STEP, NEXT statements can be heaped together into the computer, then three or more are also possible. In the case of the Apple II's 280-by-160 graphics mode, you can use this feature to create movement of images of changing colors.

In any program, a loop controls all the statements contained within that loop, including any number of other loops. You can use each loop to increase or decrease one of several variables handled within the core of commands that you want the loop to repeat. One loop can thus control one variable.

In the case of the arrows on the screen, for example, the color of each arrow can be changed before the next arrow is drawn. In particular, if you use a little trick and command that the last color be black, that arrow will disappear from the screen and the next arrow will appear by itself until it, too, is drawn over in black, disappearing from the screen to be replaced by the next. The result is an elementary form of animation.

To put the color command within the control of a loop that will change its values, you can drop it from outside the loops by removing line 20:

```
20
```

then put an HCOLOR = command inside the loops and enclose it with FOR-TO-STEP and NEXT pair of statements that will run through all the color codes from white to black, 7 to 0. These statements you can add in the following form:

```
37 FOR C = 7 TO 0 STEP -1
38 HCOLOR = C
43 NEXT C
```

With program lines 37 through 43, a new innermost loop is

formed, repeating the HPLOT command for eight values of color in the same way as the other FOR-TO-STEP and NEXT loops repeat it for five positions and four rows. FOR-TO-STEP statements can count down by using a negative STEP number, as well as up with a positive one.

The FOR-TO-STEP and NEXT commands can work their magic in numeric or text programs, as well. Wherever they're found, their function is to direct the computer to repeat the commands they enclose between them.

Another clever use of the pair is worth noting, for it adds a delay that can be useful in any program in which you would like to allow time for someone to view the program's effects. You might have already noticed that the computer, incomprehensibly fast though it is, does take a perceptible time to do things. The more complex or complicated the task, the longer it takes. Quite forgivable, really, what with all the translating and calculating involved. But knowing this, you can do more than forgive the machine. You can use this delay to your advantage, by controlling it. You can add a loop containing no real active command. And though it may have no command to repeat in such a loop, the computer will go dutifully through it, taking time to translate and process the FOR-TO-STEP and NEXT commands.

To slow the animation program, then, you can add an empty loop before the color command, like this:

```
39 FOR T = 1 TO 500: NEXT T
```

You'll notice that the STEP part of the command is optional—if you leave it off, as is done here, the computer automatically takes steps of 1.

In this program the computer pauses briefly with each color, just before switching to the next one as it runs senselessly through its empty loop. This type of FOR-TO, NEXT statement can be inserted anywhere in a program to slow or freeze advancing action.

You can use these command pairs to try differing patterns, for instance, without actually knowing the limits of the values the computer is handling in variables like X and Y. You can leave it to the computer to signal you when commands have taken it past the outer limits. Using a pair of FOR-TO-STEP, NEXT commands as a counter, you can modify the program to draw an arrow diagonally descending across the screen to the right side. In place of the

nested FOR-TO-STEP, NEXT loops that alternate position, you can put a simple counting loop and a statement that increases the X and Y values. That statement would look like this:

```
44 X=X+50: Y=Y+40
```

The two movement loops are then easily replaced by a counting loop. You can do this by eliminating the outer loop:

```
32
47
```

and replacing the inner position loop with:

```
35 FOR I=1 TO 10
45 NEXT I
```

When you run this program, the computer will draw an arrow that drops diagonally through one row after another as it moves to the right, until the limits of the screen are exceeded by values in line 44. Then the computer signals you with the message:

```
ILLEGAL QUANTITY ERROR IN 40
```

You'll avoid this message by replacing the FOR-TO statement with a value you can count off the screen, 4 (the number of arrows):

```
35 FOR I=1 TO 4
```

WORDS FROM THE COMPUTER

In some ways, the Apple II can handle words and keyboard characters as it handles numbers. It can print them, and it can take them apart and piece them together. Of course, we have our own human edge over the computer, as we each understand about 50,000 more words than the Apple II's pale vocabulary of a hundred commands. But like a blind newstand vendor who can handle magazines he can't read, the Apple II can handle thousands of words it will never appreciate.

Although the computer can treat keyboard characters—words, numbers, and other symbols—in various complex ways, it groups these symbols into only two categories. When the computer encounters words and symbols enclosed in quotation marks, it treats

that group of characters like a busload of passengers, to be carried about as you, the tour leader, direct. But characters typed without quotes are treated as commands, from which the computer takes directions, just as a cab driver picks up passengers from a street corner and takes directions from them.

Variable symbols can be used to stand in for strings of characters enclosed in quotes just as they do for numbers. The Apple II is built to recognize the equal sign as a command assigning a variable to words as well as numbers. It recognizes these strings by a "tagged" variable symbol. A variable intended for a string is marked with a $, signifying a string of characters, rather than a numeric variable. Since the computer treats spaces as characters, a group of words separated by spaces will be treated as one long word string by the Apple II.

You can command the computer to substitute a sentence (or any group of characters) wherever it encounters a string variable in a program statement:

```
10 A$ = " WORDS WITHOUT THOUGHTS NEVER TO
   HEAVEN GO."
```

You can then use the string variable, A$, in place of the sentence when you give subsequent commands:

```
20 PRINT A$
```

Encountering line 20 when this two-statement program is run, the computer prints Shakespeare's words on the screen, just as it would print a number assigned to a numeric variable.

Although strings possess none of the numeric characteristics that the computer was built to work with, they can nevertheless be objects of calculation and manipulation. Using string variables, you can order the computer to combine, cut or derive numeric values from strings. The computer treats string variables in combination, for instance, much as it treats numbers in addition. In this case, the final result will be a longer string rather than a larger number. You can add strings to each other using the plus sign (+) as a command to the computer to produce a new string, which is a combination of the strings on either side of the plus sign.

Supplying two more strings for the computer to work with in

this program, you can prepare it to add them:

```
12 B$ = " WILLIAM SHAKESPEARE "
14 C$ = " HAMLET, ACT III, SCENE 3 "
```

To direct the printing of these strings in combination you can retype line 20 as:

```
20 PRINT A$ + B$ + C$
```

and, when the program is run, the computer will respond with the display shown in Figure 4.6.

You can actually extract a numeric value from a string with a command that counts the number of characters, including spaces, within it. Such a command can be useful in a program that manipulates strings. The command that directs the computer to count characters takes the form LEN( ). In a program, you can assign a string length value to a numeric variable symbol like this, in a different verson of line 20:

```
20 A = LEN(A$)
```

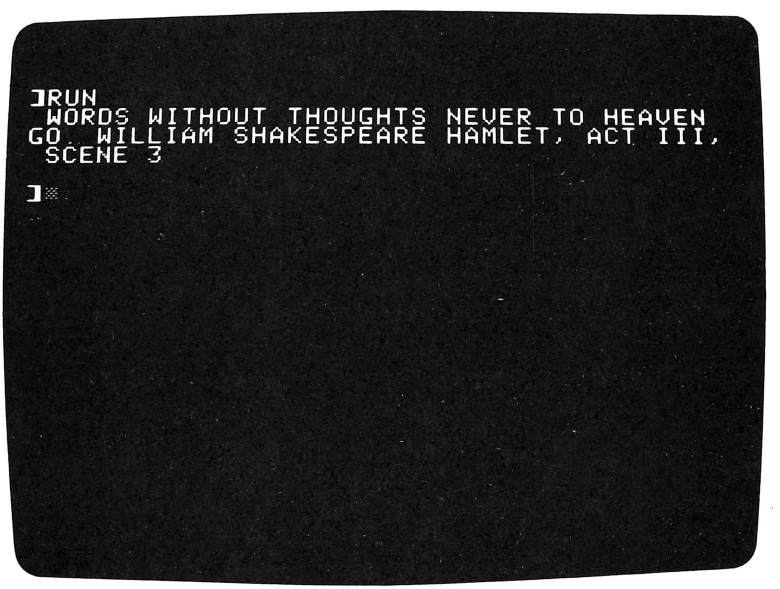


Figure 4.6: The effect of a program that adds word strings.

This directs the computer to substitute the number of characters in string A$ wherever it encounters the symbol A in a program run. You can then use this number in another program statement, like this:

```
30 PRINT " A QUOTE " A " CHARACTERS LONG BY " B$
```

and also add a statement to display the quote itself after spacing a couple of rows:

```
40 PRINT:PRINT: PRINT A$
```

Three other commands tell the computer to pick characters from within the strings. Using them, you can take apart words or groups of characters, and rebuild them according to your own preferred patterns. The three string-slicing commands are similar in form and each cuts out a segment from the string.

One of these commands, LEFT$( ), prompts the computer to slice off a section from the left of a group of characters. This command directs that the section be cut counting characters over from the left quotation mark. If you wanted a new string, called D$, formed from the first five characters of the quote here, you could name and direct its formation with the command:

```
40 D$ = LEFT$(A$,5)
```

And you could display it then with the command:

```
50 PRINT D$
```

The program, when run, would produce the display shown in Figure 4.7.

With a bit of alteration of the LEFT$( ) command, to use the length counted in program line 20 through the LEN( ) command in a FOR-TO NEXT loop, you can direct the production of new strings of every possible length from the first quote:

```
35 FOR I = 1 TO A
40 D$ = LEFT$(A$,I)
60 NEXT I
```

The program includes the string-cutting statement of line 40 and a print command at line 50 within a FOR-TO-NEXT loop that counts from 1 to the full length of the original quote. When it's run, the computer will print one row after another; each one

begins with the first character of the string and is one character longer than its predecessor. When the last cycle is reached (when I = A), D$ will be assigned the value of LEFT$(A$,A), which is the full length of A$. D$ will be the same as A$, and the last line printed will be the entire quote, as seen in Figure 4.8.

By adding another statement, you can cite the source of the quote:

```
70 PRINT C$
```

You can make this program into a test of recognition by slowing the production of each part of the quote string with an empty FOR-TO-NEXT loop, like this:

```
38 FOR T = 1 TO 500: NEXT T
```

Now, with any quote introduced by the A$ = command, you can run the program, stopping it from the keyboard with a CONTROL-C command and advancing it with the CONT command.

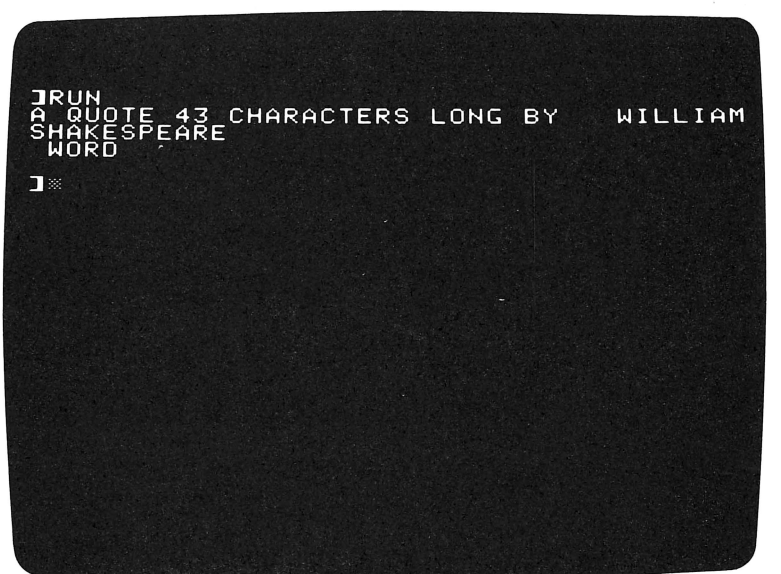


Figure 4.7: The effect of the LEFT$( )string-slicing command.

Another string-slicing command, RIGHT$( ), directs the computer to count off and slice sections of strings from the right. The program will print sections that are counted from the last letter of the last word of the string, if this right-slicing command takes the place of the left-slicing one in line 40, like this:

```
40 D$ = RIGHT$(A$,I)
```

Driven by the same loop, the computer will now print each line one character longer than its predecessor, but it will start from the right quote mark, as shown in Figure 4.9.

A third string-slicing command, MID$( ), directs the computer to snip a section from the center of a string. Using this command, you can name the string from which a section is to be excised, the leftmost character's position as counted from the left side of the string, and the number of characters to be counted and cut to the right. The command takes the form MID$(A$, left-hand edge, length of section cut). If, for instance, you were to have the eight-letter THOUGHTS cut from the quote of line 10, the excising

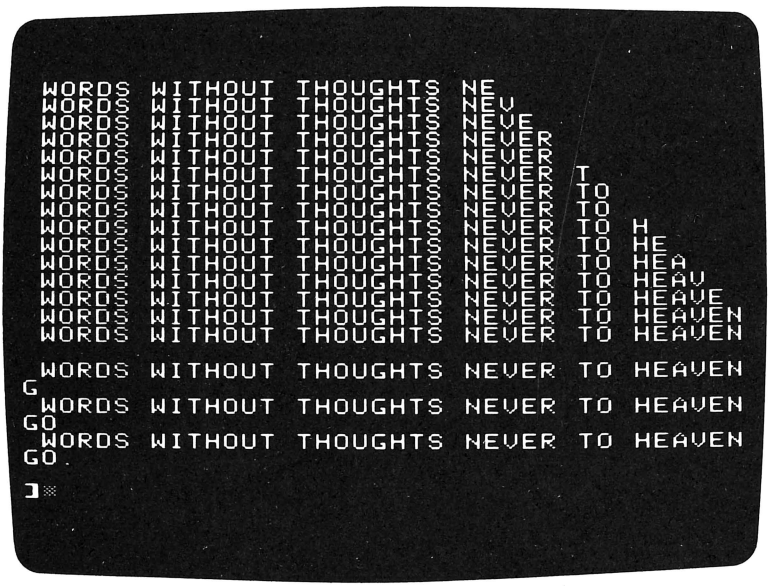


Figure 4.8: The final results of a program using the LEFT$( ) command in a program loop of lengthening slices.

command would take the form MID$(A$,16,8). You can insert this command in place of the other string cutters at line 40 to direct the computer to produce another list of strings, this time growing outward from the middle of the original string, A$, in both directions.

To use the MID$ command to produce strings growing *symmetrically* from the middle of the original string requires a twist of arithmetic, though. You can control the length of the string with the variable I, given value by the FOR-TO-NEXT statement, as it was in the LEFT$( ) and RIGHT$( ) commands.

The computer can find the middle of the A$ quote by dividing its total length (as found in line 20: A = LEN(A$)) in half: by A/2. The computer can find half the number of characters, I, to be printed at each pass, by I/2. Half of the characters will be printed from the right side, and half from the left of the center. Since the computer counts from the left beginning with a character it numbers 1, the left boundary can be specified as A/2 – I/2 + 1. You can direct the computer with the MID$( ) command to produce

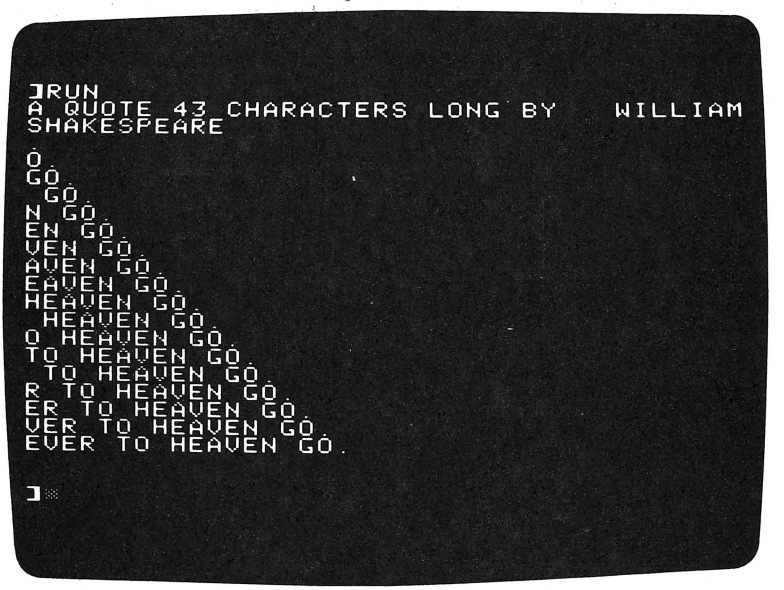


Figure 4.9: The effect of the RIGHT$( )string-slicing command in a program loop of lengthening slices.

strings from the middle of the original quote, A$, with the command at line 40:

```
40 D$ = MID$(A$, A/2 - I/2 + 1, I)
```

A list of growing strings will be printed, as in Figure 4.10.

INSIDE INFORMATION

Like versatile actors within a play, variable symbols for numbers and characters can play many changing roles within a program. The equal-sign directs these variables into their respective roles of the moment.

But the Apple II recognizes another way of assigning values to variables, somewhat the way a stage company's actors are given their assignments together. You can do this with a pair of commands, DATA and READ, which list the values sent to the computer, and the variables to which they are assigned, respectively.

Any number of values can be sent to the computer with the command that handles the information list, DATA. You can use it

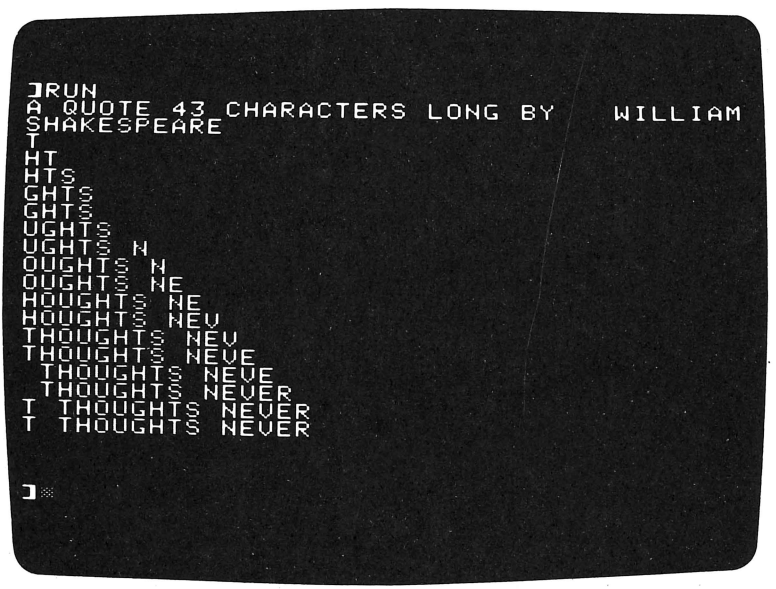


Figure 4.10: The effect of the MID$( ) string-slicing command in a program loop of lengthening slices.

to send the values of a string and two numbers, JONES and 15 and 18, like this:

```
10 DATA JONES, 15, 18
```

and then use the command that makes the assignments from the list of values, READ, to assign those values in order to the variables, N$ and A and B, like this:

```
20 READ N$, A, B
```

You can now use these assignments, which are the equivalent of the commands:

```
N$ = JONES
A = 15
B = 18
```

in a PRINT statement like this one:

```
30 PRINT N$ " LIVES NEAR THE CORNER OF " A " AND " B
```

These two commands, one forming an information list, the other drawing from that list, can be used anywhere within a program. One need not precede the other. If you place a READ statement before a DATA statement, the computer will seek out the DATA statement when it encounters the READ, and make the assignment of value to the variable.

Because it's directed to make substitutions each time it encounters a READ statement, you can command the computer to substitute several sets of values taken from a DATA statement for a single set of variables listed in its accompanying READ statement.

A program like the one just run can be modified to handle a group of values, acting on each group, then returning to the READ statement for new assignments for the variables. If, for instance, you're handling several sets of names and addresses in the form of numbered cross-street locations and you'd like to print a simple map on the screen while displaying the name and address below, you can enlarge the three-line program to do that using

PRINT and some graphics commands. You can plot the locations with these statements:

```
15 GR: COLOR = 15
40 PLOT A, B
```

Running the program as it is, you'd see a single square at column 15 and row 18, and the message

```
JONES LIVES NEAR THE CORNER OF 15 AND 18
```

as in Figure 4.11.

You can easily add more names and addresses to the DATA statement:

```
10 DATA JONES, 15, 18, FOSTER, 22,35, VINCI, 5,39,
   JACKSON, 37, 14, MORRISON, 30,16, DVORAK, 26,17,
   TUAZON, 1,17, OLIVER,34,22
```

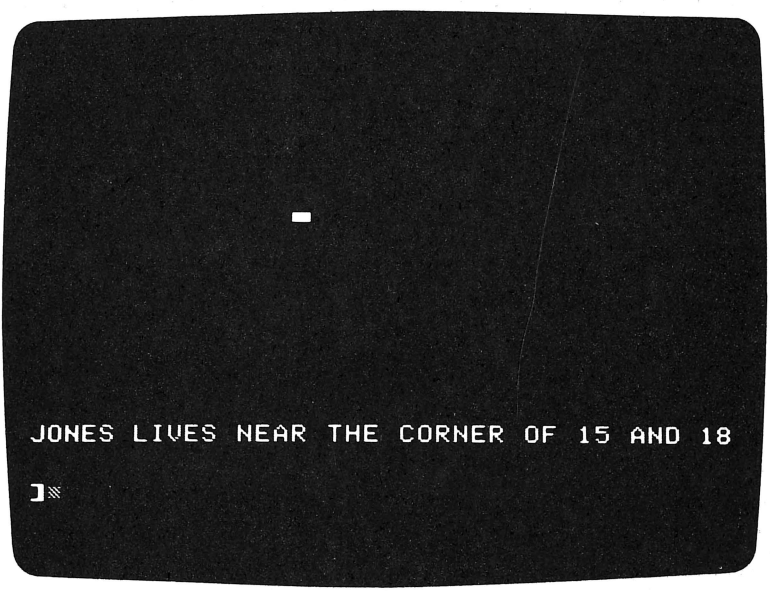


Figure 4.11: Printing and plotting in 40-by-40 graphics using DATA and READ statements.

But the computer will respond as before, printing only the information about Jones, since it acts on the READ statement only once. To direct it through the READ statement often enough to substitute each of the three-piece sets of information in the DATA statement, you can add a FOR-TO, NEXT pair of commands to repeat lines 20 through 40:

```
17 FOR I = 1 TO 7
50 NEXT I
```

This way, you direct the computer to introduce different values from the DATA statement into an often-used series of commands. To slow the plotting and printing pace, you can add an empty loop before you direct the computer to another set of values:

```
45 FOR T = 1 TO 2500: NEXT T
```

When you run this program, the computer will switch into the 40-by-40 graphics mode and prepare to plot with white. It then assigns to the variables N$, A, and B the first three values in the DATA statement, JONES, 15, and 18, respectively.

It next prints the statement from line 30 in the text window, and then draws the box plotted in line 40. After running through the empty FOR-TO, NEXT loop in line 45, the computer is next sent back from line 50 to the READ statement at line 20, which assigns to the variables N$, A and B the next three values in the DATA statement (FOSTER, 22,35), and repeats the printing, plotting, and pausing. The NEXT command in line 50 once again sends the computer back through the READ statement and another set of values.

This process continues until the last run through the FOR-TO, NEXT loop, when the last values of the DATA statement are assigned.

Besides making new assignments for variables, the DATA, READ and FOR-TO, NEXT command pairs can be useful in putting a different name, tagged by a number, on each value in a DATA statement. With a FOR-TO, NEXT loop you can direct the computer to number each variable by a pass through the loop.

To name the variables so that each matches a single value, you can change line 20 to:

```
20 READ N$(I), A(I), B(I)
```

which will number the variables. On the first pass through the loop, when the FOR-TO, NEXT statement has directed that I = 1, the three variables N$(1), A(1), and B(1) will be assigned to stand in for JONES, 15, and 18. On the second pass, when I = 2, the three variables N$(2), A(2), and B(2) will be named to represent FOSTER, 22, and 35. The computer repeats this process until the last pass, when N$(7), A(7), and B(7) are named and assigned.

You'll now have seven sets of second-named, or *subscripted,* variables that you can direct the computer to bring into a program statement by number, like this:

```
30 PRINT N$(I) " LIVES NEAR THE CORNER OF " A(I) " AND "
   B(I)
40 PLOT A(I), B(I)
```

The resulting program will produce the same results as its unsubscripted predecessor. But the advantage is that you can direct the computer to use these numerically-named variables, according to numbers calculated or assigned in other program statements.

You can, for instance, direct the tabular printing of the locations. This calls for the computer to switch to the text mode and clear the screen. You can command this, after a delay that keeps the map on the screen for a moment, with the combination of commands:

```
60 FOR T = 1 TO 2500: NEXT: TEXT: HOME
```

Then you can add an introductory statement and a blank row with:

```
70 PRINT "PEOPLE ARE COMING FROM: ": PRINT
```

Now the subscripted variables, A(I) and B(I), come in handy in a combination of PRINT and FOR-TO, NEXT statements that will display all the addresses:

```
80 FOR I = 1 TO 7
90 PRINT "AVENUE " A(I) " AND BOULEVARD " B(I)
100 NEXT I
```

The program now will direct first the mapping and printing as it

did before, then the clearing of the screen and printing of a list of locations, as shown in Figure 4.12.

If you have a use for variables with more than a single numeric tag each, you can add more tags. The variables N$(I,J), A(I,J), and B(I,J) could be named by wrapping another FOR-TO, NEXT loop that counts with J around the I loops. The numbers for J could stand for the work shifts of each person, for example, and could range from 1 to 3. You can use several loops in this way to set the computer to naming variables with the form N$(I,J,K,L,M), in which I was a code number for the person, J his shift, K the weekday, L the destination, and M the preferred route of travel from the location.

Each loop of FOR-TO, NEXT statements that you use to assign values to subscripted variable names multiplies the number of variables. Each DATA statement ought to contain as many values as requested by the accompanying READ statement. When the computer encounters a READ command asking for more information than is provided by a DATA command, it will stop the program and print the message:

```
OUT OF DATA ERROR
```

The computer, when first encountering a subscripted variable that is not numbered, automatically sets aside space for as many as eleven numerically tagged variables that might later be produced. In this way, it reserves places in its voluminous memory for each of the assignments—A(1) = , A(2) = , and so on—that your program might generate from DATA and READ commands. The eleven variables will be numbered 0 through 10, for each set of subscripts. Thus, whenever the computer encounters a variable A(I), it reserves space for the variables A(1), A(2), and so on to A(10).

If you want the computer to handle variables numbered outside this range, you can direct it to put aside space for them. With the command that orders this, DIM, you name the variable, and give the subscript the highest number you'd like to allow for. To reserve space for the variables N$(0) through N$(50), for instance, you can send the command:

```
DIM N$(50)
```

If, for instance, you're considering fifty people who can each work one of three different shifts, and you wish to reserve space for

all possible arrangements of names and cross-street addresses, the variables resulting could take any of the forms between N$(1,1), A(1,1), and B(1,1) to N$(50,3), A(50,3), and B(50,3). You can use the DIM command to save space for all of these variables and values by sending the directions:

```
DIM N$(50,3), A(50,3), B(50,3)
```

in a statement at the beginning of the program. On encountering this command the computer will arrange for an array of spaces in its memory starting with N$(0,0) and ending with N$(50,3). No space problems will befall a program that runs without using all the space set aside by a DIM statement. But the program that tries to crowd in variables without reservations will be stopped in its tracks by an intolerant computer with the reply:

```
? BAD SUBSCRIPT ERROR IN
```

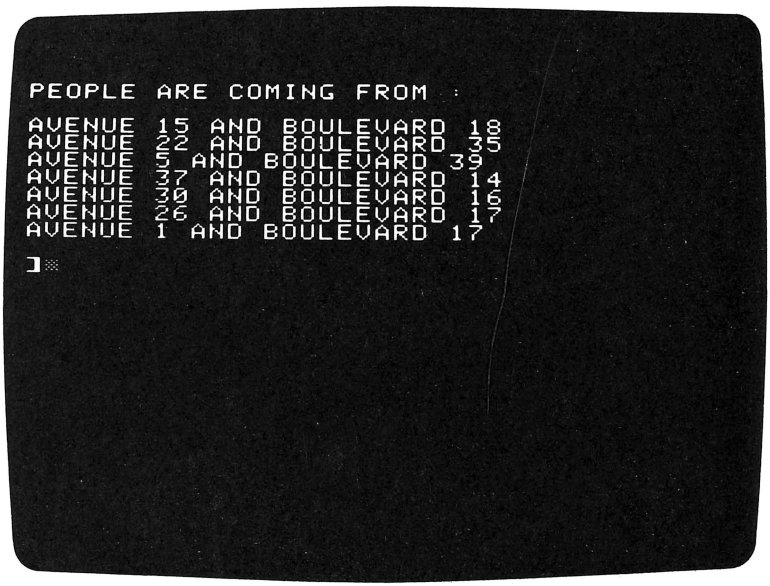

PEOPLE ARE COMING FROM :
AVENUE 15 AND BOULEVARD 18
AVENUE 22 AND BOULEVARD 35
AVENUE 5 AND BOULEVARD 39
AVENUE 37 AND BOULEVARD 14
AVENUE 30 AND BOULEVARD 16
AVENUE 26 AND BOULEVARD 17
AVENUE 1 AND BOULEVARD 17

Figure 4.12: Use of subscripted variables from one part of a program in another part to make a list.

OUTSIDE CONTACT

Even while it is working on the commands of a program, the Apple II is not entirely blind to what's happening out there where you are. It can respond to directions more subtle and informative than the throttling interruption you give it with a CONTROL-C command or by switching off the power. But the computer is unimaginative, and has to be pointed in the right direction before it can know what you have to tell it.

You can add a command to a program so that the computer will stop everything, take the information you have for it, then proceed with the run using what you provided. That command is INPUT. So commanded, the computer can interact with a person at the keyboard or with devices to which it's connected.

The INPUT command directs the computer to prime itself by setting aside a variable, or several, and then to siphon the value, or values, typed at the keyboard. It then proceeds with the program, substituting those values as though it were responding to an assignment (=) command or a DATA, READ command pair.

Using a simple form of the command, you can direct the computer to tell you it's waiting for information (and then take what you type in with a press of the RETURN key), and include it in the program. The command takes the form of a simple request for a variable's value. You can use it to direct, for example, an instant doubling of whatever number you send from the keyboard in this short program:

```
10 INPUT A
20 PRINT A*2
```

The statement, INPUT A, directs the computer to print a question mark on the screen as a signal to you that it's waiting for a reply from the keyboard. If you do nothing, the computer will remain frozen in its operation at line 10, waiting. If you type a number, say 537, and then press the RETURN key, the computer receives that number, makes the assignment A = 537, and then carries out the commands on the next line. In this case, the multiplication 537*2 is printed on the next row as 1074.

The INPUT command translates as: "Print a question mark, take the next value sent from the keyboard, and substitute it for the variable in the statements that follow." The INPUT command

can also direct the computer to act on strings in the same way, as in this short program:

```
10 INPUT A$
20 PRINT A$ + A$ + A$
```

which will stammer three times whatever you send.

Like a DATA, READ pair, a single INPUT command can be used to direct the computer to act on both strings and numbers, as in this program:

```
10 INPUT A$, A, B
20 HOME: VTAB A: HTAB B: PRINT A$
```

If you run this program, and then respond with:

```
FROM THE KEYBOARD, 20, 10
```

the computer will then clear the screen, and print the words FROM THE KEYBOARD at row 20, column 10.

You can use the INPUT command like a DATA, READ command pair, typing values after the question mark (as after a DATA command) and setting variables for assignment after the INPUT command (as after a READ command). There is a difference, however: Using an INPUT statement, you can send new values during each program run without changing a line of the program.

The INPUT command also allows you to direct the computer in a more convenient way by condensing two commands into one. This second form of the INPUT command directs printing on the screen, in place of the question mark, before waiting for a reply from the keyboard. You can use it by typing, after the word INPUT, a question or statement you'd like displayed, and then typing a semicolon to separate the variables you're directing the computer to ask for.

After sending a NEW command to clear the last program from memory, you can build a program from this altered form, starting with:

```
10 INPUT "WHAT'S THE CODE PLEASE ? ";A$
```

which directs the computer to print your quote, "WHAT'S THE CODE PLEASE ?," and then wait for a string that can be substituted for A$.

With the exception that it's more graceful in positioning, in this form the INPUT statement replaces the following commands:

```
10 PRINT "WHAT'S THE CODE PLEASE" : INPUT A$
```

With a twist of a conditional (IF-THEN) statement, you can program the computer's responses to vary according to replies from the keyboard. By arranging a program so that the computer encounters a conditional IF-THEN command after an INPUT command pulls keyboard information into the program to be used in other commands, you can make the program a two-way exchange of information, an *interactive,* computerized conversation.

To build it from the current INPUT statement, line 10, you can add an IF-THEN command and two possible responses, like this:

```
20 IF A$ = "THE ANCIENT MARINER SENT ME." THEN 40
30 PRINT: PRINT " I DON'T TALK WITH STRANGERS. ":
   GOTO 10
40 PRINT: PRINT " OK. I'LL SPILL MY GUTS TO YOU. THIS
   COMPUTER RUNS ON A 6502 CHIP. I'M JUST AN
   INTERPRETER FOR THE PROCESSOR CHIP THAT
   REALLY RUNS THE SHOW AROUND HERE. WHEN YOU
   GIVE COMMANDS IT'S THE 6502 CHIP THAT MAKES
   THINGS HAPPEN."
```

When you run a program like this, the computer is directed to take a value (or a string of characters) from the keyboard, then compare and act according to that value.

The INPUT command grabs a string sent from the keyboard and makes the assignment of A$ = . The IF-THEN command compares against the quote. If there is no match, the next command, line 30, directs a rebuff, "I DON'T TALK WITH STRANGERS.," and sends the computer to act on the INPUT statement at line 10, which again asks, "WHAT'S THE CODE PLEASE ?"

When the reply sent from the keyboard (and assigned to A$) makes the IF condition true, the computer is sent by the THEN command to line 40, to PRINT the computer's inner secrets, after which the program ends.

When reply after reply from the keyboard doesn't match the condition of the IF-THEN statement, the computer is trapped in the loop of program lines 10 through 30 through 10, printing the

rebuff and requesting the code over and over.

The Apple II is particularly stubborn with INPUT statements. Most other programs can be stopped dead in their run with the CONTROL-C command. Not so with programs waiting for a reply to an INPUT command. You can stop the program with a firmer CONTROL-RESET command, which directs the computer to stop the program run but to retain the program in its memory. You can later pick up the program at the point where it stopped, with the command CONT.

You preset the response of the computer, of course, when you write the program, but you can also make the response depend on the result of a numeric calculation or on some other value given to the processor through some external device.

There is another command, born of the same need for information as the INPUT command, that directs the computer in a similar, but terser, manner: GET. With it, you can command the Apple II to act on a key pressed at the keyboard, without waiting for the RETURN key. It directs immediate substitutions and can order values for several variables listed in it, but prints no message in quotes, or question mark. The main limitation of this command is that it takes only one character, number or string, for each variable.

You could use the command in a program of its own, like this:

```
10 GET A$
20 HOME: VTAB 12: HTAB 20: PRINT A$
```

When this short program is run through the Apple II, the computer assigns the first key pressed to the variable A$, then clears the screen, and at the center prints the character of that key. If more than a single key is pressed, as in

```
TO ERR IS HUMAN
```

the computer will take only first letter typed, T, and display it at the center of the screen, ignoring any other characters typed until the program has finished.

Add a GOTO statement to make a repeating program and you can see how, under the direction of the GET command, the computer takes just one character for each variable.

```
30 GOTO 10
```

Now, the key pressed first after the program begins will be displayed until replaced by another key press. When it is acting on a GET statement, as when acting on an INPUT statement, the computer can be freed only with a CONTROL-RESET command.

Several variables can be assigned by a single GET command, as they can by a single INPUT command. This program will wait for three keys to be pressed, after which they'll be displayed center screen:

```
10 GET A$, B$, C$
20 HOME: VTAB 12: HTAB 20: PRINT A$ + B$ + C$
30 GOTO 10
```

The program below will direct the computer to position and print a letter according to the keys pressed. The first two keys are numbers. The third key is a string and will be printed.

```
10 GET A,B,A$
20 HOME: VTAB A: HTAB B: PRINT A$
30 GOTO 10
```

Since numbers can also be considered as characters, a response of 143 will direct the computer to print the character "3" at row number 1, column number 4. If you typed something like NAP, or even 2AP, you'd receive a ?SYNTAX ERROR reply, since the computer tries to assign numbers from the first two keys pressed.

The features that make this command, GET, either useful or bothersome, depending on your viewpoint, are two: it directs the computer to ignore more than the number of variables it's set to assign, and it acts without waiting for a press of the RETURN key.

The Apple II can reach beyond the keyboard for information to act on. It can do this through its circuitry. You need not call an electrician or take out a soldering gun. In fact, if you plug in a pair of game paddles into the back of the Apple IIe (or into the chip socket inside on the motherboard; see Figure 4.13) you can open another supply line of information.

The Apple II has the ability to electronically sense the positions of the paddles or joystick, and to interpret those positions as numbers. You can direct the computer to take values from the paddle positions, for example, with a command, PDL( ), that introduces

them into a program. Then, acting as though these values had been assigned in any of the other ways already seen, the computer can treat them as it could any other value.

From each paddle, the computer can read one numeric value; from a joystick, it can read two. Using a command that translates the electrical status of a paddle or joystick into a numeric value that varies from 0 to 255, you can direct the computer to assign that value to a variable, which can then be acted on in the program.

The PDL( ) command can be used in a statement like the one below, which assigns the value of each of the paddles (or the forward-backward and left-right movements of a joystick) to the variables X and Y.

```
10 X=PDL(0): Y=PDL(1)
```

The command PDL( ) translates as: "Generate a number from the electrical value of the paddle this way—when the paddle is turned completely counterclockwise generate the value 0; when it

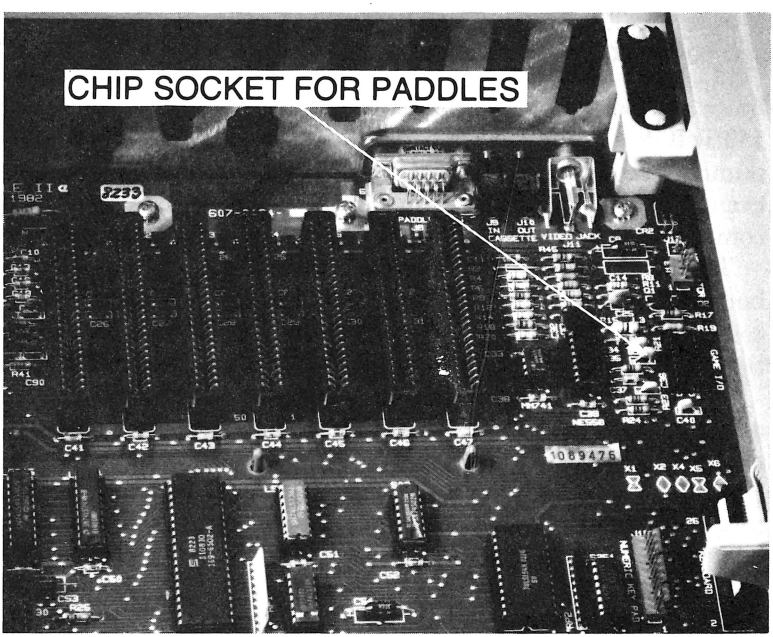


Figure 4.13: The chip socket on the motherboard where some game paddles plug in.

is clockwise, generate the value 255; generate values in between according to position."

The statement in line 10 prepares the computer to substitute the value from paddle number 0 for the variable X, and the value from paddle number 1 for variable Y. If they aren't marked, you can find out which is which by trying them with a program like the one that follows. You can add printing, screen-clearing, and positioning statements, and a repeating loop command, like this:

```
 5 HOME
20 VTAB 12: HTAB 15: PRINT X" "
30 VTAB 12: HTAB 15: PRINT Y" "
40 GOTO 10
```

When you run this program, the paddles feed in values through the PDL( ) statements. Program lines 20 and 30 each direct the computer to print the value of the position of one paddle near the center of the screen. Each PRINT statement ends with blank spaces in quotes to erase residual characters, such as the zero from 100 when the paddle is turned to position 99. If you left these spaces off, incidentally, the countdown from 100 would appear as 100, 990, 980 on the screen.

On encountering the GOTO statement, the computer is directed back to the command that assigns the paddle values to variables. Thus, the values printed on the screen keep pace with the current paddle positions.

As this program is run, what you see on the screen are values generated from a physical device acting as an extension outside the computer. That might give you some ideas.

You can use these values within a program just as you'd use values generated or supplied within it. With this program, for instance, you can send a conditional IF-THEN statement to make the game paddles into an electronic combination lock:

```
35 IF X = 222 AND Y = 157 THEN 50
```

This statement directs the computer out of the GOTO loop when the positions of the paddles are as specified. Starting at line 50 you could put anything, including a whole series of commands that do fantastic things or reveal obscured secrets. Or you could put a

statement that makes the program a sort of safecracker's test:

```
50 PRINT "YOU'VE FOUND THE COMBINATION!"
```

With two so easily manipulated values, you can set in motion all kinds of action. For instance, you can draw directly with the paddles by switching the computer into a graphics mode and deriving the dot column and row values from paddle positions. Changing the program to draw in 280-by-160 graphics, you can make these adjustments:

```
 5 HGR: HCOLOR = 15
20 HPLOT X,Y
30 GOTO 10
```

Then you can delete the other commands:

```
DEL 35, 50
```

And to keep the values from the paddles proportioned to the screen dimensions, you can rewrite the paddle assignment statement:

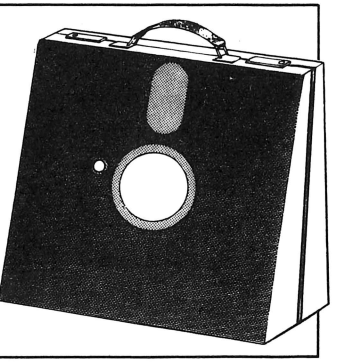
```
10 X = (279/255) * PDL(0): Y = (159/255) * PDL(1)
```

which will direct the computer to coordinate the movement of the paddle knobs over the full range of the screen without exceeding the row limits of Y.

All manner of programs can direct your Apple II. By choosing or designing them, you make the active computer a reflection of your own interests.

CHAPTER FIVE

MOBILE MEMORIES—DISKS



A MARRIAGE OF MACHINES

Like an obedient work elephant, the Apple II can carry out an enormous number of commands. Unfortunately, each new program you type in will replace or write over its predecessor, which is then mangled or lost forever by the flow of commands into the computer's memory.

The computer was built to act on the statements you send it in a coherent program, treating the whole series of commands as a single set of orders. When the power to the computer dies, so dies the program in its memory. And only by recording a program in some way is there any way to use it again.

It's likely that you're using an Apple II because—features of its pioneering history, cost-per-kilobyte or style of design aside—the thousands of programs already written for it form an enormous pool to draw from, to help you do things like keeping books or following objects in space.

How—you may ask—do I get those programs into my computer and my programs back again without numbing my fingers in an endless marathon of keyboard pounding?

Well, farms have silos, offices have file folders, and music studios have recording tape. The Apple II, like any respectable personal computer, is equally well provided for, perhaps even better. It has a *disk system.* This system will operate under the electrical and logical control of the Apple II as an extension of the Apple's

own memory, a warehouse of program commands and information. Each element of this system provides more than twice the storage capacity of the Apple IIe memory, which is 64 kilobytes. With such a system, you can put aside and use again your own programs and other programs already written for the Apple II.

Practical, affordable and fast, the disk storage system works through a book-sized box and thin plastic circles, known as *disks,* each contained loosely within square plastic envelopes.

These disks, which give their name to the system that operates from them, are firmly coated with a magnetically-sensitive material and respond like recording tape to an electromagnetic device that scans their surfaces inside the box. That box, the *disk drive,* comes complete with a motor for spinning the disks within their envelopes, phonograph-style. Since these disks can be run in a drive one at a time, and then removed and replaced by other disks, they form an interchangeable and unlimited set of repositories for programs and information passing into or out from the computer.

Figure 5.1 shows a typical disk drive and some common disks.

Once connected to the computer electrically, the disk system also responds to commands not present in the vocabulary of the Apple II. You can supply the disk operating directions for these commands from a disk itself. Through these commands, you can

Figure 5.1: Disks and a disk drive.

direct the computer to put aside on a disk a program or information from its memory, and to pull from a disk into its memory. You can tell the computer in one program to run another program from a disk.

The Apple II, which draws or deposits on one disk at a time, will accommodate six disk drives. But you can handle most disk uses conveniently with only two. In fact, a single disk drive will suffice for most disk uses, though less conveniently, at times requiring more inserting and withdrawing of disks on your part.

Between a disk system and the computer, a curious and interactive exchange of control takes place. Your commands can direct the computer to a particular disk drive (if more than one drive is installed) and to a particular program or file of information on the disk in the drive box. In converse fashion, a program directing the computer's action from a disk can guide it to carry out commands or sequences just as you would from the keyboard.

Each program or information file brought out from a disk is first fed into the Apple's memory before any action is taken on it. The three elements of the system you connect to use the disk are the computer, the disk drive, and an *interface card,* shown in Figure 5.2.

In each drive box resides the circuitry for running its motor and moving the searching magnetic drive head. The computer can control this operation and supply needed power with the help of an

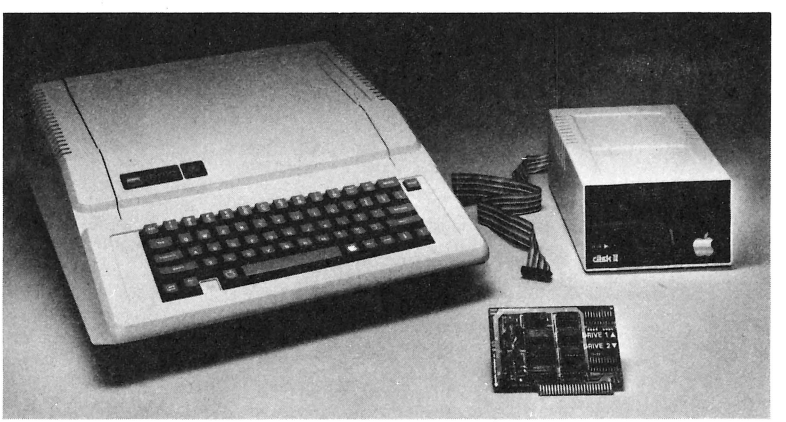


Figure 5.2: The three elements that connect to form a disk drive system: the computer, the disk drive and an interface card.

added bit of circuitry, which takes the form of a plug-in card that forms an electrical pathway for information going one way or the other.

You can install this disk controller card (Apple's version is labeled "DISK II INTERFACE CARD") in that pathway in the same way that you insert a key into a lock or plug a light cord into a wall socket. In a single ribbon of wires, coming from the back of the drive box and ending in a narrow, boxy plug, is every electrical connection needed between the drive and the computer. First, connecting the drive to the computer, the boxy plug fits onto the interface card, then the interface card fits into the computer. Each card accommodates two drive plugs and will control two drives.

You won't need an electrician to connect things. Here's what to do. First, turn off the power switch at the back. With the power off, you can't damage the machine if you do the following: Take off the Apple II's lid, by pulling up on it from the rear on either side, as in Figure 5.3. You might hear snapping as two locking snags there pull apart when you lift. Raise that rear edge to about twice the height of the computer, then slide it out and backwards, away from the keyboard end.

With the cover removed, you can look down at what resembles a town of miniature black buildings seen from the air. This is the computer's circuitry. At the left side is a large metal box, the power supply. Near it and toward the rear of the inside in the IIe sits a pea-sized red bulb that should be unlit as an indication that

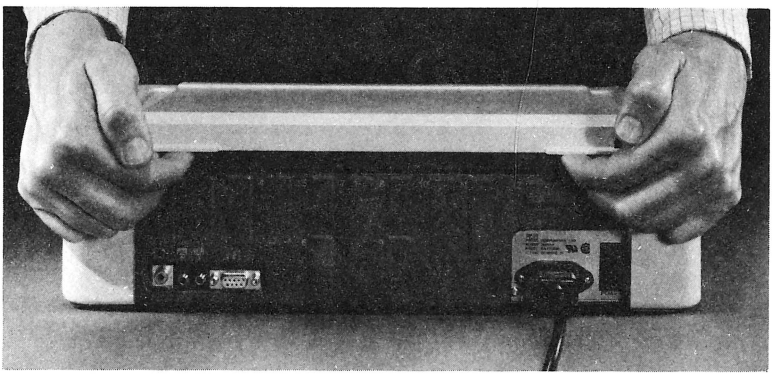


Figure 5.3: Opening the case by pulling up on the lid.

the power is turned off. As a precaution, to safeguard the circuitry you'll handle, touch the metal box to discharge any static electricity you might have picked up.

First, snake the ribbon of wires through the back of the Apple II. The metal panel at the back of the IIe is designed so that each long vertical port can accommodate one disk ribbon. Previous versions have long notches into which wires can be laid. Each port on the IIe comes plugged with a plastic insert. By tugging up the tab end of each insert and pushing it out from the inside, you can open as many ports as you'll need, one ribbon to a port.

Open the ports and snake in the drive connectors for as many drives as you're adding, starting with port 1, labeled on the outside of the back panel.

If you're adding just one or two drives you can use a single interface card. One side of the card is labeled DRIVE 1 and DRIVE 2, with small triangles pointing to the connections on it. The ribbon connectors attach there.

After you've brought the ribbon connector for each drive through the back, you can connect it to the card. To connect at DRIVE 1, hold the card so that the side with the labeled little black boxes is facing you and the words DRIVE 1 are to the right. Take the connector of the disk drive ribbon wire and, holding it so that the rows of holes on it are facing the rows of metal pins, align the holes and pins so that each pin is pointing into a hole on the connector. Check the right and left pins to be sure each will find a home in the connector. Then slide the connector down onto the pins.

If you're connecting a second drive, attach it in the same way to the pins labeled DRIVE 2 with the ribbon connector from the other drive box. With the drive box now connected to the interface card, you can connect the card to the computer.

The computer is designed to electrically look for a disk connected to the interface card in slot 6 when it's turned on. On the large printed-circuit board holding the little town of black boxes near the back of the computer is a row of plastic connecting slots (7 on the IIe, 8 on others), each about the size of a thin finger. A number on the board labels each slot at its rear end. Hold the wired card up over slot 6 so that the card's ribbon wires and boxes face right, away from the metal power supply box. With the card oriented so that the printing faces up and its tab end of short silver

strips faces downward into the teeth of the slot, you can rock the card from end to end while pressing firmly but gently down, until the tab is clamped by the slot's teeth.

The electrical connection between computer and disk drive system is now complete. On the IIe, you can do a bit of tidying up if you like, by slipping each of the ribbons trailing out of the computer into one of the U-shaped clamps provided. The clamp fits around the ribbon and into the port on the back panel. Screw it down to hold the ribbon in place.

To close the Apple II, slip the front end of the lid back into place behind the keyboard while holding the rear end up. Slide the lid so that it lines up with the contours, and snap it into place by pushing down on the rear edge. You can position the parts of the system—computer, disk drives and monitor or television—wherever the wires will reach.

USING PRERECORDED DISKS

Now electrically in control of the disk drives, the computer still needs a working vocabulary of disk operating directions so that your commands will move information and programs between disks and the Apple. These directions are encoded on the prerecorded disk sold with the disk drive. More about this later, but first, a word about how to treat disks.

Disks, like paper, come in two forms, blank and written on. You can copy recorded disks, write on blank parts of recorded disks, and write over or erase parts of recorded disks. You can also protect the contents of a disk as you might put written paper in a binder so that it won't be unexpectedly written on. The square envelope around the disk protects the magnetic writing from erasure or alteration in the drive if that envelope is completely sealed around all edges and lacks a cut-out notch. If such a notch (called a "write-enable" notch) is exposed on an envelope, as in Figure 5.4, the disk drive will be switched out of a reading-only operation and you can direct the computer to put your typing onto the disk.

You can freeze from alteration all the programs and information on a disk bearing such a notch, by covering that notch with a sturdy piece of foil tape, which is usually provided with blank

disks. When the mechanisms of the disk drive sense that this space is covered, the drive will be switched to read-only.

As you might have guessed from their nearly total enclosure, disks are fairly fragile themselves. Even when a disk appears to have endured some mishap, its magnetically-sensitive surface can be damaged by magnetic fields, extremes of temperature (those outside our comfort range), bending or folding, and contamination by dirt, dust, skin oils, and the like. Information and programs recorded on the disk surface in invisible magnetic patterns can be cripplingly altered by an imperceptible scratch or contamination.

Ensconced in its envelope, the bulk of the disk escapes the hazards of contamination at any one time. You can handle the envelope without concern. But where the disk surface itself peeks

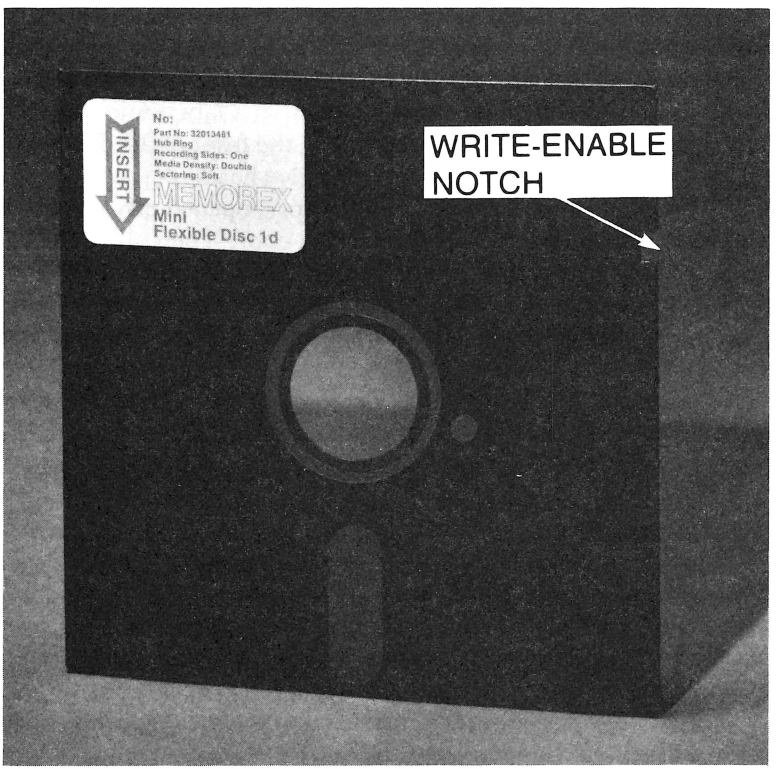


Figure 5.4: Disk with write-enable notch exposed.

through an oval slot, the drive head reads. And through that window can pass dust and contaminants as well.

You can easily avoid common disk problems in two ways, by removing disks from the drive before switching off computer power and by marking labels before sticking them onto envelopes.

The disk, in its plastic envelope, is stored in a paper jacket which doesn't quite cover the envelope. This paper sleeve can shield the otherwise exposed disk surface at the oval cutout when it's out of the drive box.

The magnetic head in the drive box, recording and reading along a short track, makes contact with the disk through the envelope slot while the disk itself is spinning inside the envelope. Because the disk envelope is square and two-sided, there are eight possible ways to put it into the drive. Only one way will position the envelope and readable side of the disk by the drive head.

The drive head works at the rear half of the drive box, so the disk envelope slot goes in first. The disk has one unlabeled side on which the ends of the plastic envelope are crimped in an unfinished-looking way and another, smooth side that's labeled. If this finished, labeled side faces the little door at the front of the drive, the magnetic side of the disk will face the drive head.

So you can insert the disk into an empty, opened drive box like this: slot in first, labeled side facing the door. Then all you do is slide the disk gently in until it passes the face of the drive box opening and you can close the door over the center of the opening so that it's flush with the face of the drive box. You can remove the disk by hooking a finger under the door to open it, and then sliding the disk out. A disk inserted in a drive box behind the closed door physically completes the computer-disk system. But without directions to act on disk commands, the system is about as useful as an illiterate librarian. These directions on how to manage the disk come from a special disk itself.

Even when physical and electrically connected with a disk system, the computer must still be told what to do with this apparatus, like a puppy facing for the first time a stick thrown by its master: until the dog is directed in this new activity, its master can't command it to retrieve.

Apple II computers are designed to take disk-operation directions from a disk itself. But how—you might ask—can the computer take

instructions on operating a device from that device when it doesn't have directions for operating the device in the first place? There are two ways out of this paradox. Each directs the Apple to connector slot 6 where it exchanges information through the connection marked DRIVE 1 on the card there. In fact, the Apple II does this automatically each time it's turned on either with the power switch, or by the keyboard command, CONTROL/HOLLOW-APPLE/RESET.

If a disk drive system is installed, the computer sends directions through the drive #1 connector of the controller card to start the drive and search a preplanned section of the disk for operating directions. If those directions are found, the computer reads them through the drive head and stores them in a reserved part of its memory, just as it stores your programs from the keyboard. Thereafter, when the computer encounters a command referring to disk drive operation, it will read the directions in that part of its memory and carry out the command.

You can supply this set of directions from the disk marked "DISK OPERATING SYSTEM MASTER," the DOS disk. Besides other useful and demonstrative programs, this disk holds the system of disk operating directions that make the physical parts work.

Once loaded into its reserved section of memory, the DOS directions—about 20 additional commands for Apple DOS—are held aside from any program statements you may type into the computer. The Apple II is designed so that, no matter whether programs come and go, or are obliterated with the command NEW, these disk directions remain until the Apple II power is turned off.

If, in searching slot 6 for these directions when it is turned on, the computer finds an empty drive or only a blank disk, the drive motor will spin unstopped, the screen will remain blank except for the Apple II title, and the computer will no longer respond to any command from the keyboard but one. At this stage it's still looking for direction. If you insert a disk with only DOS disk directions and close the drive door, the computer will eagerly put those directions in its memory, turn off the drive and print a flashing cursor as a signal of its readiness. Then, you can type in programs or commands and the computer will respond to them.

If you want to stop the automatic spinning without adding DOS directions, you can regain control with a CONTROL-RESET keypress command from the keyboard. The computer will turn off the drive motor and electrically and logically ignore the presence of the disk drive and interface card, and instead will present its ready right-bracket prompt and the flashing cursor, as a signal of its obedience.

Then, if you later want the DOS directions taken into the computer, you can send the command:

```
PR#6
```

This code tells the computer to search slot number 6 inside and to act on the interface card that's installed there, just as it does automatically when you turn the power on.

When the Apple II automatically takes disk operating directions into its memory from a disk, it also receives commands to automatically erase any program in its memory, to put there a specially marked program from the disk, and to RUN that program. The DOS disk supplied with the drive system gives the computer such a program, the effects of which are not fully apparent on the screen. The DOS disk also holds other prerecorded programs you can call into running with disk commands. (DOS disks labeled January 1, 1983 or later contain programs similar to those described in this chapter, but some are under different names and in somewhat different arrangement. The functions of the MAKE TEXT and RETRIEVE TEXT programs, for instance, may appear on a second disk labeled SAMPLE PROGRAMS under the names MAKE TEXT and GET TEXT, respectively. These later programs differ in detail from their predecessors, but function similarly and can be altered in like manner.) If you insert the disk into the drive that starts up, drive #1 of slot #6, this is what will happen when you turn on the Apple II.

After the disk system operating directions are quickly loaded into the computer's memory, the DOS disk's first program clears the screen and prints two lines of titles, and then a line in parentheses that cryptically tells you something else is being loaded.

The message

```
(LOADING INTEGER INTO LANGUAGE CARD)
```

is the computer's signal to you that it's storing Integer BASIC, another version of the BASIC programming language that you've been using, in another reserved part of its memory. The DOS disk designers programmed the disk that way because early versions of the Apple II were built with that earlier form of the BASIC language. As a result, many older prerecorded programs available on disk are written in that form, not the form known as Applesoft BASIC, which is built into the IIe and II + versions. When it has fully taken in this form of BASIC, the computer displays the flashing cursor as a signal of its readiness for commands or programs.

Since DOS directions were loaded into the memory in the first of this three-part process, the computer can respond to your disk commands, as well as the Applesoft BASIC commands built into it. The disk directions automatically compile a listing of programs and file information as they are put onto the disk. The DOS disk, holding several programs already, has room for more. You can command a listing of the programs on a disk, and the computer will respond with a display screen showing the total capacity of the disk and a listing of the programs and files. Each is preceded by a code letter that indicates the language, version or nature of the listing: "A" for program in Applesoft BASIC language, "I" for a program in the earlier Integer BASIC, "B" for a program in the machine's binary numeric language, and "T" for a file of information. Also preceding each listed name is a number indicating how much disk room the program or information file occupies.

You can direct the computer to display this list by sending the command:

CATALOG

The screen will fill with a list like that shown in Figure 5.5. A CATALOG listing can have more items than will fit on a screen. You can see subsequent screenfuls if you press any key except RESET. The computer will display a screenful, then wait for another key to be pressed. When it displays the final item of the CATALOG list, the computer signals its readiness for your next command with the flashing cursor and right bracket mark.

If you look at this CATALOG listing, first on the list you'll see the name of the program the computer runs automatically when you turn on the Apple II. The greeting program that directs the

blanking of the screen, the display of a two-line title, and the loading of Integer BASIC into the computer's memory, is named HELLO on the CATALOG list.

The Apple DOS CATALOG takes two screenfuls to display. When you use the CATALOG command to look at a list, the program currently in the computer's memory will be unaltered. You can even stop a program, check the catalog listing, and then return the computer to the program with a CONT command.

The computer is built to act on the command, RUN, by carrying out the statements of the program in memory. But DOS directions, once loaded into the Apple II, give the RUN command a new twist. You can send the command to RUN a particular program listed on a disk's CATALOG. The computer will respond by first searching the disk for a program with the name you give. If it finds that program, it puts it into memory and then runs it from memory.

For instance, you can run the first program of the DOS disk, with the command:

```
RUN HELLO
```

You will, in this case, see the same sequence of title and parenthetical statement (LOADING INTEGER INTO LANGUAGE CARD) that appears when you turn power on with a disk waiting in drive 1.

You can run any program from a disk, but the computer follows commands to run programs in different languages or computer dialects in different ways. When you send the command RUN for a disk program in Integer BASIC—one marked with an I—first the computer signals by changing the bracket prompt to a > prompt, and the checkerboard cursor on a IIe to a solid one. After running an Integer program like ANIMALS, you can switch back to the Applesoft BASIC language mode with the command:

```
FP
```

You can, incidentally, switch to the Integer mode with the following command:

```
INT
```

Any time Integer has been loaded into the computer, you can

switch back and forth between these two language versions.

The other type of program listed on the DOS disk, the B, or binary-language program, is written in the numeric language of the computer's processor. Fortunately, you can run programs written in this language without learning it. To run the program named BOOT13, which works with disks from an earlier version of DOS, you would send the command:

BRUN BOOT13

The command BRUN is interpreted by the computer through its DOS directions as: "Run this binary language program." (You can stop the run of a program like BOOT 13, which will be of little use unless you're using an earlier disk, by the keypress command CONTROL-RESET.)

You can also direct the computer to bring any program from the disk into its own memory without running it. To bring the program HELLO into the memory, use the command:

LOAD HELLO

```
]CATALOG

DISK VOLUME 254

*A 006 HELLO
*I 018 ANIMALS
*T 003 APPLE PROMS
*I 006 APPLESOFT
*I 026 APPLEVISION
*I 017 BIORHYTHM
*B 010 BOOT13
*A 006 BRIAN'S THEME
*B 003 CHAIN
*I 009 COLOR DEMO
*A 009 COLOR DEMOSOFT
*I 009 COPY
*B 003 COPY.OBJ0
*A 009 COPYA
*A 010 EXEC DEMO
*B 020 FID
*B 050 FPBASIC
*B 050 INTBASIC
```

Figure 5.5: The first screen of the CATALOG listing of the DOS disk.

The LOAD command directs the computer to erase its program memory and to copy onto it a program from the disk. It's like typing NEW and a program. The binary version of this command is BLOAD.

The LOAD, RUN, and CATALOG commands are all you need to use prerecorded disks operating under the directions of Apple DOS.

All commands that direct the computer in using a disk carry the unspoken assumption that the drive in use is attached to the DRIVE 1 connection of the interface board in slot 6. The computer automatically sends directions that control drive 1 of any installed interface card and prepares to receive information and programs from it. You can direct the computer to any other disk or slot by adding that disk or slot number to any disk operating command. If, for instance, you'd like the computer to run the program HELLO from the drive connected to DRIVE 2 of the interface card in slot 6, you could give this command:

```
RUN HELLO, D2
```

The command D2 tells the computer to expect information from drive 2 of slot 6, instead of the usual drive 1.

Likewise, you can direct the computer to operate a disk drive connected to an interface card in a slot other than number 6. If you've installed an interface card in slot 5, for instance, and you wish the computer to operate the disk in drive 1 of that connection, you can run the program HELLO from there with the command:

```
RUN HELLO, S5
```

Since the drive is unspecified here, the computer assumes that input will be from drive 1. The same command, altered so as to run the program from drive 2 of slot 5, would take the form:

```
RUN HELLO, S5,D2
```

Connecting a disk drive to anything but the expected drive 1 of an interface card in slot 6 presents no particular advantage or problem, except in the case of a prerecorded disk that directs the computer to slot 6, drive 1. A program with such directions will fail to work from another drive or through another slot.

PROGRAMS FROM COMPUTER TO DISK

The DOS disk already contains many programs, but like an unfilled sheet of paper that's been written on, it still offers space. As you collect programs you'd like to keep, you can use this space or space on other disks for those programs.

Disks are fragile and subject to losses of small, but significant pieces of information the more they are handled. It therefore makes sense to make duplicates of each expensive or hard-to-replace disk. Disks can be used with relative confidence, but as time and use accumulate, so do the chances of some damage to a piece of information on that disk. To duplicate a disk like the DOS disk, you'll need a blank disk (a single-sided, single-density one will do) and less than five minutes using a program, COPYA, from the DOS disk itself.

Run this program and the computer will carry on a cryptic little dialogue with you through the screen and keyboard. In the terminology of the program, the disk in the ORIGINAL slot and drive will be duplicated onto the disk in the DUPLICATE slot and drive. If your interface card was installed in slot 6 and you have two disk drives, then you can advance the copy instructions by pressing the RETURN key after each line appears on the screen. This automatically directs the computer to the original in slot 6, drive 1 and the blank in drive 2. The computer program, lapsing into computerese, calls these "default" values.

If your disk system arrangement is different, you can type the slot and drive numbers that describe it. If you have only one disk drive in the system, press the "1" key after the word DRIVE appears under duplicate. Your system will read and copy through the same drive. Instructions for handling the disks will appear on the screen.

With an open-notched disk in hand—such as a duplicate of DOS disk made according to the COPYA program with the original safely tucked away from surprise dust and magnetic storms—you can direct the computer to copy your keyboard typing or the contents of its memory onto that disk.

The computer, in preparing for a greeting program as DOS directions are loaded into its reserved memory, automatically erases its program memory. So it's naturally a forward-thinking

idea to put the computer through its disk DOS-loading-and-greeting-program-run ritual before typing in a program you want kept on the disk.

You'll find it simplest to move programs to drive 1 even if you have more drives, because the computer automatically looks for a system disk there.

You can record a program on the disk, by first typing it into the computer. Any program will do—the Hamlet quote from Chapter 4, or one of your own invention.

To clear the program memory of the disk's greeting program, you can send the command:

NEW

Next you can type in, copying or devising a new one, the program you'd like to keep on disk, just as you would before running it. You can, of course, run it to see that the computer does what you want with the program. When you've got the statements as you like them in memory, everything's ready for the disk command that will preserve them outside the computer.

All that remains is for you to name the program. The disk directions tell the computer to handle names up to 30 characters long. If you want to call your program COLD STORAGE, for example, you can direct the computer to store it under that name by the command:

SAVE COLD STORAGE

The disk drive will whirr and cluck and make brief, fabric-tearing sounds. When the commotion stops and the drive light goes off, the program exists in two places. Magnetically encoded, it's on the disk. And as before, it's in the computer's program memory. It hasn't moved; it's been copied.

If you type LIST, the program statements will appear on the screen. You can type NEW to erase the program memory and still run the program, this time from the disk, with the command:

RUN COLD STORAGE.

It will also be freshly loaded into the program memory of the computer, of course. You can remove the disk, turn off the computer, send the disk round-trip to Chicago, for instance, and—if it

returns—put that disk in the drive box, turn on the computer and, after the greeting ritual, type:

RUN COLD STORAGE

and the program will manifest itself.

The computer was quietly following another disk direction when you commanded SAVE COLD STORAGE. It put the name of the program, preceded by its language type and the space it takes, on the CATALOG list. If you now send the command:

CATALOG

and advance it to the end, you'll see:

A (a number like 002) COLD STORAGE

You can change the name of a program with another disk command, RENAME. Directed by this command, the computer looks for the program under the first name listed, then substitutes for it the second name.

To change the name of the program COLD STORAGE to STOWAWAY, you can send the command:

RENAME COLD STORAGE, STOWAWAY

So directed by the RENAME command, the computer changes only that program's name, leaving the program itself and its position in the CATALOG unaffected.

Also, there's no need to clutter a disk with a program you might no longer want. To erase a program once and for all, you can give a command like this:

DELETE STOWAWAY

The computer, under DOS directions, will operate the drive to find the program, erase it entirely, and remove its name from the CATALOG listing.

You can move programs individually from one disk to another using the LOAD, DELETE, and SAVE commands. First you LOAD the program from one disk into the computer's memory, then DELETE that program from the disk on which it originated, and then SAVE it by sending it from the program memory to another disk.

To move a program like STOWAWAY using a one-drive system, you can first bring it into the computer's program memory:

```
LOAD STOWAWAY
```

have it erased from the disk with the command:

```
DELETE STOWAWAY
```

then remove that disk from the drive, insert a second disk—one equipped with its own set of disk directions, like another copy of the DOS disk—into the drive and send the command:

```
SAVE STOWAWAY
```

To move a program using a two-drive system, with the program on the disk in drive 1, for instance, and its destination on the disk in drive 2, you can give this sequence of commands, waiting for the disk commotion to stop between commands:

```
LOAD STOWAWAY, D1
SAVE STOWAWAY, D2
```

(At this point the program exists in three places: on both disks and in the computer's memory.)

```
DELETE STOWAWAY, D1
```

You can, through the SAVE, DELETE, LOAD and RENAME commands, juggle and reshape programs on notched disks quite freely. But your power over disks is not yet complete. Disk programs can be marked with the LOCK command to stop the computer before it carries out any command that would alter a disk program. One difficulty that can be side-stepped in this way is the natural human tendency to SAVE a program on disk under a name already used there. Without a special halt mark, the computer would write the new program in place of the earlier one, whose statements would be lost.

The programs you find on the prerecorded DOS disk were all marked so that your DELETE, SAVE or RENAME commands won't alter them. This special status is noted by an asterisk preceding each program in the CATALOG listing.

To prevent alteration of a program like STOWAWAY, you can command:

```
LOCK STOWAWAY
```

You'll then see in the CATALOG listing an asterisk to the left of the STOWAWAY entry.

This tag inhibits the computer from carrying out commands that would change disk programs or files. You can remove it from a program so protected by sending the complementary command, in the case of STOWAWAY:

```
UNLOCK STOWAWAY
```

You can put programs and information on any disk that has a set of disk directions and a magnetically-etched pattern. This etching is automatically directed by the COPYA program of the DOS disk. A disk in the drive without these directions is as useless to you with the computer as a bare desk to a writer with a pen. With them, the disk is like a blank sheet of lined paper, laid ready on the desktop.

Any disk with a set of disk directions can be used to duplicate those directions and a magnetically-etched pattern of sectors onto another disk. That second disk, so supplied, can then be used to supply other disks.

You can use a single command, INIT, to add disk directions and sectors onto a blank disk and, in the same effort, copy a program from the computer's memory as the first automatically-appearing program on that disk. When disk directions are in the computer's memory, you can command the computer to so prepare a blank disk with disk directions and a magnetic format of its own. The first step is to put the program you'd like as a greeting into the computer's memory. One like this one-liner would suffice:

```
10 HOME: PRINT "THIS DISKETTE IS INITIALIZED AND
   READY."
```

The next step is to be sure the disk is in the drive. Then you can direct, with the same command that will copy disk directions, the etching of the magnetic format and copying of the program. If you call the above one-line program START, the command that will

set things in motion is:

INIT START

The computer will set about controlling the disk drive, and you'll hear the fabric-tearing sound of magnetic formatting for several moments. When all is quiet again and the disk drive light goes off, the newly-initialized disk is at your service.

If you start up the computer by switching on the power or with the CONTROL/HOLLOW-APPLE/RESET comand with this disk in the drive, the computer will take in disk directions into its reserved memory and run the first program. In this case, that program clears the screen and puts there the message:

THIS DISKETTE IS INITIALIZED AND READY.

The newly-prepared disk can hold program and information files just as the DOS system master disk can. You can selectively put useful programs from another Apple II disk (like the DOS disk) on any initialized diskette.

You could copy a program like MAKE TEXT, for instance, with the command:

LOAD MAKE TEXT

Then, replacing the DOS disk with an initialized one that you'd like to hold the program MAKE TEXT, you could command:

SAVE MAKE TEXT

And the computer would duplicate that program on the disk you had initialized.

Using a two-drive system, you could insert the initialized disk in drive 2 and give the command:

SAVE MAKE TEXT, D2

with the same results. The computer would then be switched into a drive 2 mode.

Using this procedure you can duplicate selectively and move programs around your diskettes as you find useful. In particular, you can copy programs like the information- and file-handling programs of MAKE TEXT and RETRIEVE TEXT onto a disk that will be used for a special purpose, like holding information files.

FACTS AND FILES

Programs are useful enough, but let's face it, what you really want the computer to do is handle information—data, facts, figures, names, addresses, times, places, quantities, descriptions.

On a disk you can actually put any information you can type. You can store it under a name on the CATALOG listing and bring it out through the computer to the screen. You can direct the computer to work on it, manipulating according to program statements. Like files in a cabinet, information on a disk is at your disposal to pull out for reading, or to use in programs in place of DATA and INPUT statements. However, the computer needs more detailed directions for putting information that's not organized as a program on disk, and for pulling that information out back to its memory.

Among the programs of the DOS disk are two that handle information files on disk. One program directs the computer to put keyboard information on an initialized disk, the other directs it to read information from a disk. More elaborate programs that direct the computer in manipulation of information files in complex ways are commercially available on prerecorded disks. If so inclined, you can write some yourself by building from the essence of the two DOS disk programs. They each provide a set of commands that direct the computer in information transfers.

In carrying information files—also known as *text files*—to and from a disk, the Apple II treats them rather blindly, just as it treats strings of words in a program. Whether you direct numbers, words, commands, or declarations of emotion into these text files, the computer will treat them only as a group of characters.

You can use these two file programs without studying how they work. You can also list these programs and strip away all but the essential file features to fashion a streamlined sequence of file-handling statements that you can fit to other programs for manipulating information and perhaps making new files, if you like.

The program that directs the computer in forming information files is listed on the DOS disk as MAKE TEXT. When you send the command:

```
RUN MAKE TEXT
```

the computer clears the screen and prints a three-line description of the program's purpose, then directs you to type the first item you'd like included in the text file. After you send the first typed element by pressing the RETURN key, the computer prints a prompt on the screen requesting the second item. This process continues until you press the RETURN key twice without pressing a character key. In reply to this signal that the list is complete, the computer signals its readiness for a file name for these items.

If, in response to the MAKE TEXT program requests, you type the items ALMAVIVA, DRACULA, and MONTE CRISTO, and then name the list COUNTS, the computer will respond with the following display.

You'll see a command that has an effect like opening a file folder of the name you've given:

```
OPEN COUNTS
```

Then a command shakes the file clean of any contents that may have been filed under that name before:

```
DELETE COUNTS
```

Then one opens the file for the information you've sent:

```
OPEN COUNTS
```

Then a command sends the items to be stored in the disk file:

```
WRITE COUNTS
```

Then the total number of items counted from your entries is printed:

```
3
```

And next the file items just copied onto the disk are printed:

```
ALMAVIVA
DRACULA
MONTE CRISTO
```

A final command directs the computer away from the file now on disk:

```
CLOSE COUNTS
```

You'll find, after running the MAKE TEXT program, that the computer has added the file to the end of the CATALOG listing under the code T, for text file. The COUNTS file listing would appear as:

```
T 002 COUNTS
```

You can use a program that complements the MAKE TEXT program to bring into the computer, and onto the screen, a file so formed. It's on the DOS disk under the name RETRIEVE TEXT. To use it you command:

```
RUN RETRIEVE TEXT
```

If you do, you'll see a heading on the screen. Whoever wrote this program decided to include the message that, as with the MAKE TEXT program, the same tracing of the disk commands has been switched on. And the commands are printed on the screen as they are carried out. The command that directs this, MON, does so in the same way as the TRACE command directs the line number printing of a statement being acted on during a program run. It's here revealed by the cryptic line:

```
MON C,I,O IS IN EFFECT
```

Next the computer signals that it's prepared to search for the text file:

```
NAME OF TEXT FILE?
```

After you answer, the rest of the program is run by the computer in much the same manner as the MAKE TEXT program. You can see that first the computer opens the file:

```
OPEN COUNTS
```

and copies the disk information into its memory:

```
READ COUNTS
```

As commanded by this retrieval program, the computer then prints the number of items it has counted and lists them, like this:

```
?3
?ALMAVIVA
?DRACULA
```

```
?MONTE CRISTO
```

The computer is directed away from the file, as displayed in the command:

```
CLOSE COUNTS
```

And finally, the tracing of commands is switched off and so indicated with the message:

```
NOMON C,I,O
```

The file information has been copied item-by-item into the computer from the disk, as though in response to keyboard typing after an INPUT statement, and still is present on the disk.

If you're using an initialized disk on which you'd like to store the two text file programs, you can LOAD them one at a time into the computer's memory, and use the SAVE command to put them on the disk for convenience.

Both programs, MAKE TEXT and RETRIEVE TEXT, use disk commands to control the computer from disk directions. You can add your own program statements to those of each program to direct the computer in either generating and then storing a list of items, or in taking action on a list it pulls from a disk.

If, for instance, you want the computer to display the file items on the screen in a particular way, you can add the statements to do this to the end of the RETRIEVE TEXT program. To use the file items as called by that program, the added statements need only be consistent in the variable names given in them.

To find those names you can display the program statements:

```
LIST
```

as shown in Figure 5.6.

In the Apple II DOS RETRIEVE TEXT program, the items from a disk file are actually dealt with by an INPUT statement that reads the items as they're pulled from the disk. The variable names given to the items pulled through the program are found within that INPUT statement. In this case, that name is found in line 70 of the RETRIEVE TEXT program:

```
70 :INPUT A$(J)
```

(The colon is unnecessary, probably left over from a larger statement by a programmer who didn't clean up after himself.) In the statement, you see that the computer labels the items of the disk in the form A$(J). You can use those items after that point in the program by referring to them in your added statements in that form.

So to clear the screen and print out the items neatly spaced you can add these program lines:

```
110 HOME
120 FOR J = 1 TO I
130 PRINT: PRINT A$(J)
140 NEXT J
```

This altered form of the RETRIEVE TEXT program now resides in the computer's memory. If you command the computer to run it, a descriptive heading of three lines will appear as before, and the computer will print the question:

```
NAME OF TEXT FILE?
```

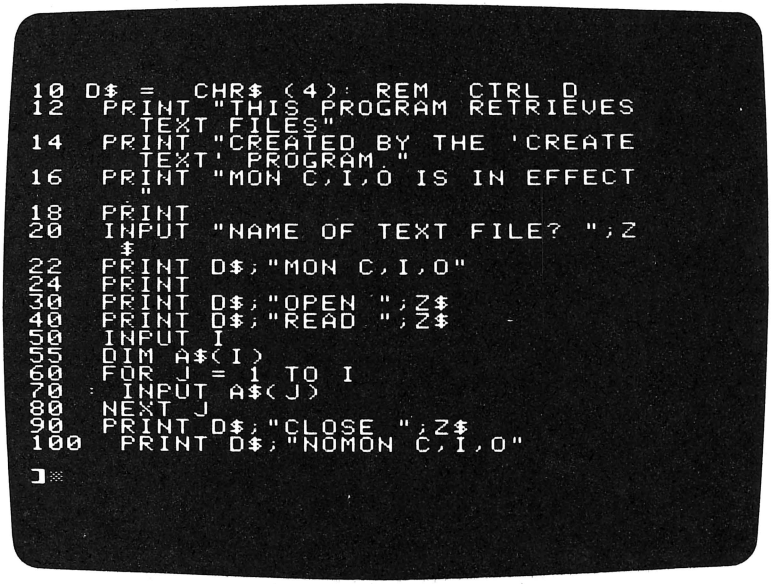

10 D$ = CHR$ (4): REM CTRL D
12 PRINT "THIS PROGRAM RETRIEVES TEXT FILES"
14 PRINT "CREATED BY THE 'CREATE TEXT' PROGRAM."
16 PRINT "MON C,I,O IS IN EFFECT"
18 PRINT
20 INPUT "NAME OF TEXT FILE? ";Z$
22 PRINT D$;"MON C,I,O"
24 PRINT
30 PRINT D$;"OPEN ";Z$
40 PRINT D$;"READ ";Z$
50 INPUT I
55 DIM A$(I)
60 FOR J = 1 TO I
70 INPUT A$(J)
80 NEXT J
90 PRINT D$;"CLOSE ";Z$
100 PRINT D$;"NOMON C,I,O"

*Figure 5.6: The listing of statements in the **RETRIEVE TEXT** program of the **DOS** disk.*

If you answer COUNTS, for instance, the three items will be printed on the screen first, as before. Then the screen will clear and the names will appear more neatly spaced, according to program lines 110 through 140.

By borrowing a program, in this case RETRIEVE TEXT, from a prerecorded disk, you can thus form a new program that makes use of the programming effort put into the original disk program.

You can further tailor one of these file-handling programs by cutting off the excess program material from the orignial. Consider, for example, the program lines up to number 20.

Line 10 is essential in that it defines a variable using a control key command, CHR$(4), that directs the computer from a program in memory to the disk directions:

```
10 D$ = CHR$(4): REM CNTRL D
```

Statements of PRINT CHR$(4) or equivalent follow the same commands given by pressing down the CONTROL and D keys, which sends the computer to disk directions in reserved memory. Line 10 is therefore crucial to the program in operating the disk drive and circuitry.

But the following lines, 12, 14, 16, and 18, are descriptions you might not want displayed, so feel free to delete them with:

```
DEL 12,18
```

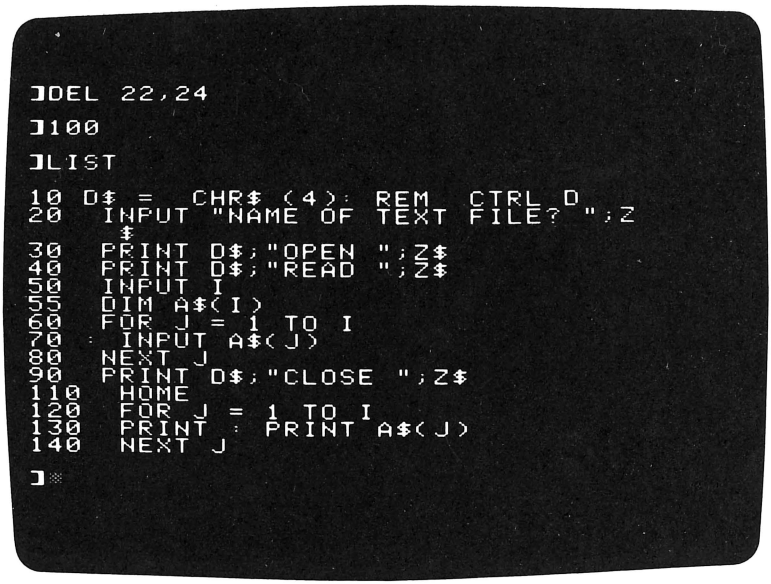
Program line 20 asks for the file name, and as such is obviously vital. Line 22 switches on a mode of tracing disk commands and information as exchanged, and unless you're searching for problems within a disk program, you don't need it or line 24, which is a spacing line. So you might delete them both:

```
DEL 22,24
```

Lines 30 through 90 form the core of the RETRIEVE TEXT program, the essentials that pull files. Line 100 simply switches off the tracing mode switched on in line 22 and is needless by itself; you may wish to delete it with an empty statement:

```
100
```

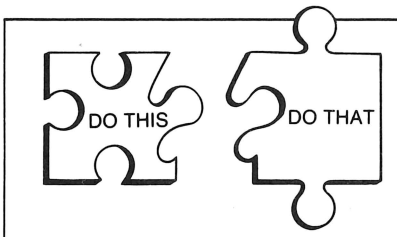
The resulting program you'll see on commanding LIST (as shown in Figure 5.7) is lean and to the point. This program will direct the computer to ask you for the name of a file which is then



```
]DEL 22,24
]100
]LIST
10  D$ =  CHR$ (4): REM  CTRL D
20   INPUT "NAME OF TEXT FILE? ";Z
     $
30   PRINT D$;"OPEN ";Z$
40   PRINT D$;"READ ";Z$
50   INPUT I
55   DIM A$(I)
60   FOR J = 1 TO I
70   : INPUT A$(J)
80   NEXT J
90   PRINT D$;"CLOSE ";Z$
110   HOME
120   FOR J = 1 TO I
130   PRINT : PRINT A$(J)
140   NEXT J
]
```

*Figure 5.7: The listing of a revised **RETRIEVE TEXT** program.*

retrieved and printed neatly spaced. You can add any statements you desire after line 90, so long as they take into account the form of the variables in the INPUT and allied statements, in which I stands for the total number count of items received, J the number of each item, and A$(J), the stand-in variable name for the item as it comes from the file. Conversely, you can change these variables to match your program if you like. This approach to program tailoring will also be of use with the MAKE TEXT program, if you wish the computer to store on disk information it generates or receives within a program. You can, in fact, streamline and tailor any prerecorded program the computer will LIST, by eliminating statements you consider superfluous.



CHAPTER SIX

CONTROL COMBINATIONS

ADDING PROGRAM PARTS

All original work, it has been said, comes from piecing together earlier work according to new creative plans. Computer use is no different. In fact, you can direct the Apple II to piece together programs (or parts of programs) you find particularly useful or interesting. You can add program parts within the computer's memory or add disk programs in succession. You can do this understanding little more of the programs or program parts than how they connect. You can create your own style of programming that calls on ready-made, often-used sequences. Using this approach, you can take statement sequences you have confidence in and splice them into a program you design to take advantage of each part.

For instance, you can direct the computer to display a name at the bottom center of the screen with a two-line program like this:

```
 10 DATA FREDDY THE WEASEL
1000 HOME: VTAB 23: HTAB 15: READ A$: PRINT A$
```

You can use the screen-clearing, positioning and printing features of line 1000 again by directing the computer to carry out the commands of that line for other names (i.e., other values of A$). The pair of commands, GOSUB and RETURN, that will direct the

computer to line 1000 and back can be added to the program, like this:

```
 20 GOSUB 1000
1010 RETURN
```

A third statement, END, is needed to stop the computer from acting on line 1000 again. It can be placed anywhere before line 1000, like this:

```
100 END
```

On encountering the GOSUB command, the computer skips any intervening lines of statements and acts on line 1000 immediately, and then, in normal sequence, performs the commands of any lines that follow that. When the computer encounters the command RETURN, it's directed back toward the GOSUB statement and the line immediately after it. Having carried out the commands between line 1000 and the RETURN statement, the computer resumes the sequence at line 100, which directs it to stop.

The GOSUB command translates as: "Mark the line number of this statement in your memory, then carry out the commands beginning at the line number indicated." The RETURN command translates as: "Return to the marker made by the GOSUB command that sent you, and then continue action with the next line number following it."

Using this pair of commands, you can direct the computer to a group of statements anywhere in the program. The GOSUB command can be used several times in a program to direct the computer to a group of statements, as shown below:

```
 10 DATA FREDDY THE WEASEL
 20 GOSUB 1000
 30 DATA JACKIE THE JUMPER
 40 GOSUB 1000
 50 DATA RALPH THE STALWART
 60 GOSUB 1000
100 END
```

At each place it appears, the GOSUB 1000 command directs the computer to act on line 1000 and any lines that follow it. The RETURN statement directs it back to the main sequence of

DATA and GOSUB statements. If you run this program, switching with a

```
SPEED = 10
```

command to slow down the printing, you'll see the name given in each DATA statement printed at the bottom center of the screen one after another.

You can direct the computer to act on any number of command groups—known as *subroutines*—in any order. You can add a graphing function to this program with another group of commands:

```
2000 HGR: HCOLOR = 7
2010 FOR I = 1 TO 6
2020 READ A(I): A(0) = 0
2030 X(I) = I * (279/6): Y(I) = 159 - A(I)
2040 HPLOT X(I - 1), Y(I - 1) TO X(I),Y(I)
2050 NEXT I
2060 RETURN
```

This group of commands will direct the computer to switch into the graphics mode and to READ values from a DATA statement, and then, by a little arithmetic manipulation of X and Y values, to plot the lines between those values.

You can now make further use of the GOSUB command. If you expand the DATA statements in lines 10, 30 and 50, you can put the commands starting at line 2000 to work on those values:

```
10 DATA FREDDY THE WEASEL, 5.5, 23, 56, 78, 80, 99
30 DATA JACKIE THE JUMPER, 6, 12, 34, 54, 65, 77
50 DATA RALPH THE STALWART, 4.5, 16, 23, 41, 50, 103
```

These values can stand for yearly weights, or productivity, or size of vocabulary, for instance.

To direct the computer to the graphing subroutine you can add, after each of the GOSUB 1000 printing commands, a GOSUB 2000 command:

```
25 GOSUB 2000
45 GOSUB 2000
65 GOSUB 2000
```

When run, this program will direct the computer first to print the name, then to plot the corresponding set of numbers.

Because it directs the computer to another line number, a GOSUB command can be used within one subroutine to direct the computer to another group. In such a case, the computer will pass through the first group of commands to the second. Encountering the RETURN command of the second group, it will be sent back to the first group. When it encounters the RETURN command of the first group, it will be sent back to the main program sequence.

Using the GOSUB-RETURN command pair in this way, you can add subtleties and complexities to programs in a simple, "modular" fashion. For example, you can enhance the graphing feature of this program with the addition of grid lines against which to compare each of the graphs. A sequence of commands that will do this could be grouped as a subroutine, starting at line 3000:

```
3000 FOR X = 0 TO 279 STEP(279/6)
3010 FOR Y = 0 TO 159 STEP 10
3020 HPLOT 0,Y TO 279,Y
3030 NEXT Y
3040 HPLOT X,0 TO X,159
3050 NEXT X
3060 RETURN
```

You can direct the computer to draw this grid just after it's switched into the graphics mode in the subroutine beginning at line 2000:

```
2005 GOSUB 3000
```

When you run this program, first the screen will clear and the name "FREDDY THE WEASEL" will appear at the bottom. Then the computer will draw grid lines. Next it will plot the values from the first DATA statement, as shown in Figure 6.1. Then the screen will clear again, and the process will begin for the values from the next DATA statement, and so on until each of the sets of values from the three DATA statements has been handled and the computer stops at the last display.

The computer acts on GOSUB and RETURN commands in pairs. If it encounters a RETURN statement without having first encountered a GOSUB statement, it will follow its built-in

directions to signal you with a message,

```
RETURN WITHOUT GOSUB ERROR IN
```

Such a problem would result from the computer's usual progress through higher line numbers to a high-numbered subroutine. You can avoid it if you put an END or STOP statement at the end of the main sequence before the subroutine statements; in this case, line 100 END.

Like a GOTO command, a GOSUB command directs the computer to the line number that forms part of the command. You can use a variation of the GOSUB-RETURN command pair to conditionally direct the computer in and out of the main sequence of statements. As the ON-GOTO command directs the computer to deviate from the normal sequence, so the ON-GOSUB command directs the computer to particular subroutines on certain numeric conditions. You can use this command to direct the computer toward one of several command groups, from which it will be sent back to resume the main sequence again.

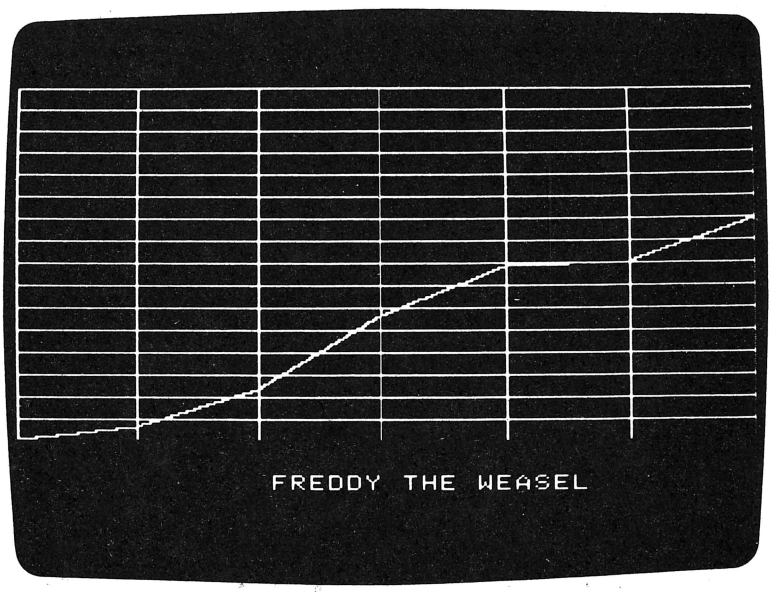


Figure 6.1: A display produced from three subroutine groups of commands.

In the plotting program, for instance, you can provide choices for the grid pattern against which the weights of these three characters are graphed. One group of commands can direct the computer to draw vertical lines:

```
4000 FOR X = 0 TO 279 STEP (279/6)
4010 HPLOT X,0 TO X,159
4020 NEXT X
4030 RETURN
```

and another group can direct the drawing of horizontal lines:

```
5000 FOR Y = 0 TO 159 STEP 10
5010 HPLOT 0,Y TO 279,Y
5020 NEXT Y
5030 RETURN
```

By changing program line 2005 to a statement whose directions to the computer are conditional, you can, according to the choice made during each program run, either use or pass over the subroutines beginning at 3000, 4000, or 5000. The command to send the computer to those subroutines, on certain conditions of the variable D, is:

```
2005 ON D GOSUB 3000, 4000, 5000
```

which directs the computer like the ON-GOTO command, comparing the value of the variable with the ON-GOSUB command's built-in ranges of value: one or greater but less than two, and so on. You can provide the value of D by replying to the computer's request in the added statement:

```
2004 INPUT "PRESS THE NUMBER OF THE DISPLAY
     YOU'D LIKE: 1 - FULL GRID; 2 - VERTICAL;
     3 - HORIZONTAL.";D
```

Running the program, you'll see the INPUT request for type of display at the window below a blank graphics screen. If a choice in the range 1, 2, or 3 is sent, the computer will first draw the graph lines according to choice, then plot the values from the DATA statement. If a choice outside the range is typed, the computer will simply move to the next program line and no graph lines will be printed before the DATA values are plotted on an unlined screen.

You can build a library of useful subroutines, numbering each

of them with high line numbers to avoid interference with a lower-numbered main sequence. In fact, your main program can sometimes be nothing more than a list of requests for subroutines, along with, perhaps, some INPUT or DATA sequences.

In assembling such a program, the only elements that must match are variables that are common to the subroutines and are acted on in each subroutine. You can be sure of this match by checking that a given variable name stands for the same quantity wherever it is used. The main sequence of such a program could be as simple as:

```
 10 GOSUB 1000
 20 GOSUB 2000
 30 GOSUB 3000
    .
    .
    .
100 END
```

The final statement, END, stops the computer from running on to the subroutines that follow. Following this structure you can take advantage of work you or someone else has already done writing subroutines, to build programs giving you greater control over your computer. You can concentrate on arranging existing elements, rather than on writing detailed program commands.

COMBINING DISK PROGRAMS

Programs exist on disks as separate, unconnected groups of commands, each beginning and ending with itself. You can, however, run a group of programs, using a program such as MAKE TEXT from the DOS disk and a command that directs the computer to LOAD and RUN one program after another from a disk.

The trick is to use the MAKE TEXT program to create a text file that directs the computer to carry out a list of RUN commands as though each were typed from the keyboard. Since you can, from the keyboard, command any program on a disk to be run, you can order any set of RUN (or LOAD or LIST) commands through this file.

You can also tag a disk program so that the computer will, on completing its run, automatically run another disk program listed in that tag. Two short programs show this program-manipulation technique. If you had one program:

```
10 HOME: VTAB 12: HTAB 10
20 PRINT " A PROGRAM CALLED EVEN "
```

stored on disk under the name EVEN:

```
SAVE EVEN
```

and another:

```
10 HOME: VTAB 12: HTAB 10
20 INVERSE: PRINT " A PROGRAM CALLED ODD "
30 NORMAL
```

stored on the same disk by the name ODD:

```
SAVE ODD
```

you could run one after the other by adding a statement to one program that sends the computer to disk directions in memory to run the other program.

If you add such a statement to the program named EVEN, the computer can be directed to run the program called ODD from the end of that first program. The statement you can add to EVEN to do this is:

```
30 PRINT CHR$(4); "RUN ODD"
```

If you now type the disk command:

```
RUN EVEN
```

the first program will run, clearing and positioning at the center of the screen the words A PROGRAM CALLED EVEN. Then automatically, the computer will load and run the program, ODD, clearing, then positioning at the screen center the words in dark-on-white letters, A PROGRAM CALLED ODD.

With the tail, PRINT CHR$(4); "RUN ODD" , pinned onto its end, the program EVEN will direct the computer to run the second program on reaching that statement. Any program on the disk connected to the Apple II can be summoned up this way.

The statement in quotation marks directs the computer as though

it were typed from the keyboard. You can form a linked set of programs by pinning such command tails on one program after another. You can even produce a loop of programs, each one taking its place in the computer's memory through endless cycles.

Using these two short display programs, you could form a loop by adding the trailing statement to the program ODD.

```
40 PRINT CHR$(4); "RUN EVEN"
```

In this case, whichever program, EVEN or ODD, is run from the disk prints its display and then directs the computer to run the other from the disk. This cycle continues as the computer operates the drives to load and run the program EVEN after ODD after EVEN after ODD, and so on, until you stop it by sending the CONTROL/HOLLOW-APPLE/RESET keyboard command, or by turning the power off. You can have the computer run a program on a disk in one drive from a program on a disk in another drive. If these programs were on different disks, the statement lines to do this might be:

```
30 PRINT CHR$(4); "RUN ODD, D2"
```

in the EVEN program, and

```
40 PRINT CHR$(4); "RUN EVEN, D1"
```

in the ODD program.

The second method you can use to run a series of disk programs doesn't rely on tagging each program, but rather on making a list of RUN commands. With this approach, you can use a program like MAKE TEXT to create a file of RUN commands. Then, if you give a disk command directing the computer, it will read that file into its memory as though you had typed it in, one item after another.

If you delete the last line of each of the two programs, first EVEN:

```
30
```

then ODD:

```
40
```

you can use this second method to string the programs together. First, you can RUN the program MAKE TEXT and, at the first

item requested, type the command:

```
RUN EVEN
```

which will be stored, like any other string of characters, in a text file. Then at the second request, you can type:

```
RUN ODD
```

You can direct the computer to send this two-item list onto a disk as a text file that you name EVEN ODD. As simply a file, this list would be displayed by the RETRIEVE TEXT program. But you can use this file with another disk command, which will direct the computer to read each item in the file as though it were sent, command-fashion, from the keyboard. That command is EXEC. To direct the computer to treat the two items of the EVEN ODD file that way, you can send the disk command:

```
EXEC EVEN ODD
```

Each file item, RUN EVEN and RUN ODD, enters the computer as though it were sent by the RETURN key, and so directs the computer first to RUN the program EVEN, then to RUN the program ODD. Incidentally, a program left in memory during use of the EXEC command will remain there even after the other two programs have been run. With the file system of running disk programs, you can combine programs sequentially, as you like.

You could run the MAKE TEXT program to make a file of these items as follows, for instance:

```
RUN EVEN
RUN ODD
RUN ODD
RUN EVEN
RUN EVEN
```

and call it, perhaps, EVEN ODD COMBINATION. The computer will run each program in the order you set in the file, when commanded to:

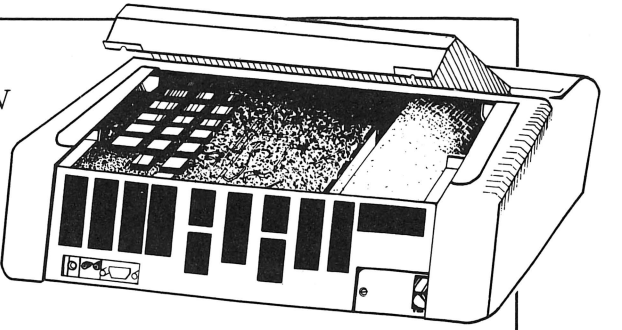
```
EXEC EVEN ODD COMBINATION
```

You can set up a file of disk commands for any collection of programs to be run in any order or number of times, if those programs

are on the same disk where that file appears. You can also use the other method, tagging RUN commands onto the ends of programs called up by the EXEC command, to form combinations within combinations. There is no limit to the interweavings that are possible. The computer will follow your directions by command, by program, and by programs that themselves direct other programs.

CHAPTER SEVEN

THE INSIDE STORY



There's control, and there's more control. If you have come through the earlier chapters, you have seen nearly all the fundamental ways to control the Apple II. Still, the story is unfinished. You can take it as far as you like.

Three more commands will lead you into the labyrinth of number codes that direct the computer. An assortment of extra circuitry will make the Apple II into something more than the machine it was built to be. And a look under the lid will reveal the guts of a machine whose "ancestors" have made computing personal.

A STEP PAST THE TRANSLATOR

The computer's *translator* acts as a bridge between you and the *processor,* which directs the flow of information through the computer's circuits. The processor and the information reservoirs (the "memory chips") sit quite apart from each other in the computer. The computer's memory has been organized so that each piece of information is kept in it according to a number, which represents the location of that data. Although the processor may change the information in a location, that numbered location continues to exist, regardless of the value contained in it, a fixed address with changing tenants.

Although it expects commands in Applesoft BASIC, the computer can be directed to internal addresses, too. You can alter the

computer's automatic responses by changing numeric values in these locations. One command will direct values into memory locations, circumventing the usual translating process.

Developing a mastery of the computer's own numeric language is a study, demanding and tedious, to which other books are devoted. Unless you have a serious passion for giving your computer directions that go beyond language commands like those in BASIC, it's a study not recommended.

The computer's numeric language is also expressed in a vocabulary of character codes of numbers and letters. These codes can represent every possible memory location and value in the usable memory of the Apple II. In this numeric language, you can send the computer to a particular memory location, like FC58 for example, where a set of directions for the HOME command tell the computer to clear the screen and position the cursor. The first location of these coded directions for HOME takes the form:

```
BD 8D C0 8E C0 30 7C
```

Seemingly endless lists of these encoded directions can blur the vision and dull the wits of all but the most determined enthusiasts.

That line of code stands for a number transcribed into a form known as *hexadecimal.*

There is one command available in Applesoft BASIC that uses the more familiar decimal numbers. You can use it to direct the computer in changing its automatic responses to others that you can specify. This command is POKE, which sends the computer to a numbered location, where the numeric value you direct is changed.

To see how the command can be useful, consider an example of changing the screen display. The Apple II prints characters in an area of the screen of a certain size and position. This screen window is determined by values copied into the active memory from the built-in unchanging memory every time you turn power on. By changing the values in these active memory locations, you can change the printing pattern on the screen. One location sets the upper edge of this print window, one the lower edge, one the left, and one the width, counted in characters from that left edge. The computer automatically prints in a text window of 40 columns by 24 rows. Counting by columns from zero on the left to 39 on the right, and by rows from zero at the top to 23 at the bottom, the

computer follows its own built-in directions in setting that window. By changing those directions within this 40-by-24 grid, you can form windows of any size.

To do this, you can use the POKE command, which slips past the translator to talk more directly with the processor. You can command the processor to set a new ceiling over the text window by directing it to put another value in memory location 34, where the screen-height value of 0 is automatically stored. If you want printing to appear at row 5 and below, you can use the command:

```
POKE 34, 5
```

This POKE command translates as: "Seek out memory location number 34, and store the value of 5 at that address."

Now, when the cursor enters the window at row 5 or below, it is restrained by the processor from appearing above that row. Printing that appeared above that line before the instruction was given will remain, unaffected by later PRINT statements. If you command HOME, for instance, the computer clears only the screen from row 5 and below, and moves the cursor to the left side of that row. As you type, the processor will limit characters printed to row 5 and below, according to its altered directions. As the lines of characters scroll up from the bottom of the screen, they disappear above row 5. Until directed by another value in memory location 34, the computer will treat row 5 as the upper limit.

You can likewise direct the processor to set another row, say number 15, as the bottom edge. The processor looks for directions for this lower limit at memory address number 35. You can use the command:

```
POKE 35, 15
```

The left-edge directions are kept in address number 32. Using the POKE command, you can set it at column 12:

```
POKE 32, 12
```

Finally, the window's width is stored in address number 33. To set it at 11 characters across, you can command:

```
POKE 33, 11
```

In combination, these four commands and the values within them (upper limit 5, lower limit 15, left side 12, and width 11)

direct the computer to position all subsequent printing in a window at the middle of the screen. When acting automatically with its normal values (upper limit 0, lower limit 23, left side 0, and width 40), the processor sends text across the entire screen.

With these commands you can effectively seize control from the built-in automatic directions, and so switch the computer into a mode of operation tailored to your special needs. Any of the computer's 65,536 available memory locations will be opened by the POKE command for your new values.

As a custom replanning tool, the POKE command in a program gives you an absolute control, directing the computer to carry out changes before advancing to the next statement line.

You can use another command, PEEK, to look at the value in any memory location. It will direct the computer to seek the numbered address you have given, as the POKE command does, and then to examine the value there, and leave it unchanged. For instance, to display the value in memory address 33, the text window width, you can send the following statement:

```
PRINT PEEK(33)
```

This combination of commands directs the computer in two stages, one command after the other. The PEEK ( ) command sends the processor to find the value, which the PRINT command then displays. If you've previously changed the value in location 33 with an earlier POKE 33, 11 command, the computer will reply with

```
11
```

the current window width of 11 characters.

Every time the computer's power is turned on, the value of 40 is automatically sent to location 33 of the active memory to set a full screen width. If you sent the PRINT PEEK ( ) statement above at any time before changing that location with a POKE command, the computer would reply with

```
40
```

You will find the PEEK ( ) command most useful if you become familiar with the memory locations used by the processor to store directions. As the computer is sent commands to carry out, the values in memory locations change. You can use the PEEK ( )

command to see what those values are at any point or to use them to further direct the computer.

You can use these memory-invading commands, POKE and PEEK, within programs. POKE statements can, for instance, direct text-window displays in blocks or overlays, according to dimensions you give. As part of a program in which the values of upper edge, bottom edge, left edge, and window width are assigned the variables U, B, L, and W respectively, POKE statements can be used in a sequence to contain text that is printed by other lines in the program. Here is such a subroutine:

```
1000 REM - BOUNDARIES OF TEXT WINDOW -
1010 POKE 34, U
1020 POKE 35, B
1030 POKE 32, L
1040 POKE 33, W
1050 HOME
1060 RETURN
```

Statements that generate text for this framed area might appear in a main program sequence, as follows:

```
100 INPUT U,B,L,W
110 GOSUB 1000
120 PRINT " THE EYES ARE THE WINDOWS OF THE SOUL."
130 END
```

Another command, CALL, sends the computer to carry out the commands whose directions begin at specific memory locations. In fact, the directions that make each Applesoft BASIC command work are loaded into such locations. For instance, since the directions for the command HOME begin at location -936, you can send the computer to follow those directions with the command that turns control over to it:

```
CALL -936
```

Using a CALL command, you can send the computer to directions at any location in its memory. At some of these are automatically stored the little numeric subroutines that serve as directions for other BASIC commands. You can put your own machine-language commands at others. If, with a CALL statement, you send the computer to the middle of one of these sequences of

directions, the results may be mangled. But by gaining some familiarity with these locations, you can avoid such surprises.

WHAT ARE THE APPLE II'S?

The Apple IIe is the most recent refinement of a computer first assembled in 1977, and then sold widely as the first of the world's *personal* computers—those an individual could buy, set up and operate independently. As the third stage of an evolution in abilities, the IIe version does more with fewer parts than its predecessors, the original Apple II and the Apple II + . According to the engineering rule of thumb that "the fewer the parts and connections, the greater the reliability," the IIe should prove to be the most reliable of its family tree.

The two most recent Apple II's, the IIe and the II + , differ in ways both hidden and apparent. Most noticeably, the keyboard of the IIe more closely resembles that of a typical office typewriter. It has features like a shift-lock key (labeled CAPS LOCK) that, like most word-processor computers, raises only the alphabetic characters to uppercase, and a DELETE key in place of a BACK SPACE key. Also, on the IIe four cursor direction keys sit at the bottom row to the right of the space bar. (On the II + , pressing ESC-D or ESC-C moves the cursor up or down, respectively.) The IIe also has keys for brackets, a vertical line, and foreign-language tilde and accent marks, but the II + does not. Finally, the IIe keyboard has the HOLLOW-APPLE and SOLID-APPLE keys discussed in Chapter 1.

There are differences at the back of the computer, too. On the IIe the handy slip-in notches for cables and ribbon wires in the II + molded plastic case wires have been replaced with 12 numbered ports in a metal rear panel said to provide radio frequency shielding for unwary home electronics from the Apple II's 11-watt powerhouse. Also, next to the cassette input and output jack, a 9-pin paddle control connector jack has been added to the IIe. That version still holds the empty socket inside, to which game controls can be connected, as the II + did.

Inside are more changes, as shown in Figures 7.1 and 7.2. The IIe microcircuitry covers more electronic turf in fewer physical pieces than the II + . On the IIe the chips, encased in black plastic boxes, number 31 on the large green "motherboard." On the

II +, the equivalent chips number over 100, including a plug-in language card that's required to measure up to the IIe.

During the time between the design of the II + and that of the IIe, circuit-making progressed, crowding more electronic circuits onto fewer chips. Six of the chips in the area bounded by a silver box marked RAM on the IIe hold the same number of circuits as 24 chips on the II +. The IIe is equipped with 65,536—64K—changeable memory locations on the eight chips in that box. The II + had 49,152—48K—such locations on its 24 chips. The 16,384—16K—additional on the IIe is organized so as to take the functional place of a plug-in circuit card with 16K of memory, on which directions for command languages other than the Applesoft BASIC can be concurrently stored.

Other chips outside the box marked RAM on the motherboard are preprogrammed, with values built into their memory locations to hold instructions in eternal readiness.

If you look at one of the largest chips in either version, you'll see the number 6502, which designates the processor of the computer. This chip, internal foreman of the others, is the feature that unites the Apple II and other computers in its class, as part of the same wide-ranging family.

The IIe and II + differ in physical construction in two ways that affect the connections you can make to other devices and circuitry. The IIe holds an upward-pointing plug of eleven pins in a row for connection to a numeric keypad, a separate connection not present on the II +.

The IIe also holds on its motherboard a slot apart from the row at the back, near the large metal power supply. This connector slot, with 60 sets of metal teeth to grip a 60-pin card—the others are 50-pin connector slots—accommodates a circuitry card that directs the computer's printing into 80 instead of 40 columns of characters across a video monitor. The II + takes a 50-pin version of this card.

Missing from the IIe and present in the II + is a connector slot number 0, into which fits the 16K memory card that held language directions other than those for the built-in Applesoft BASIC.

Both versions respond to the same set of Applesoft commands, whose directions reside in the unchanging memory chips. Programs written with these commands will work equally well on each

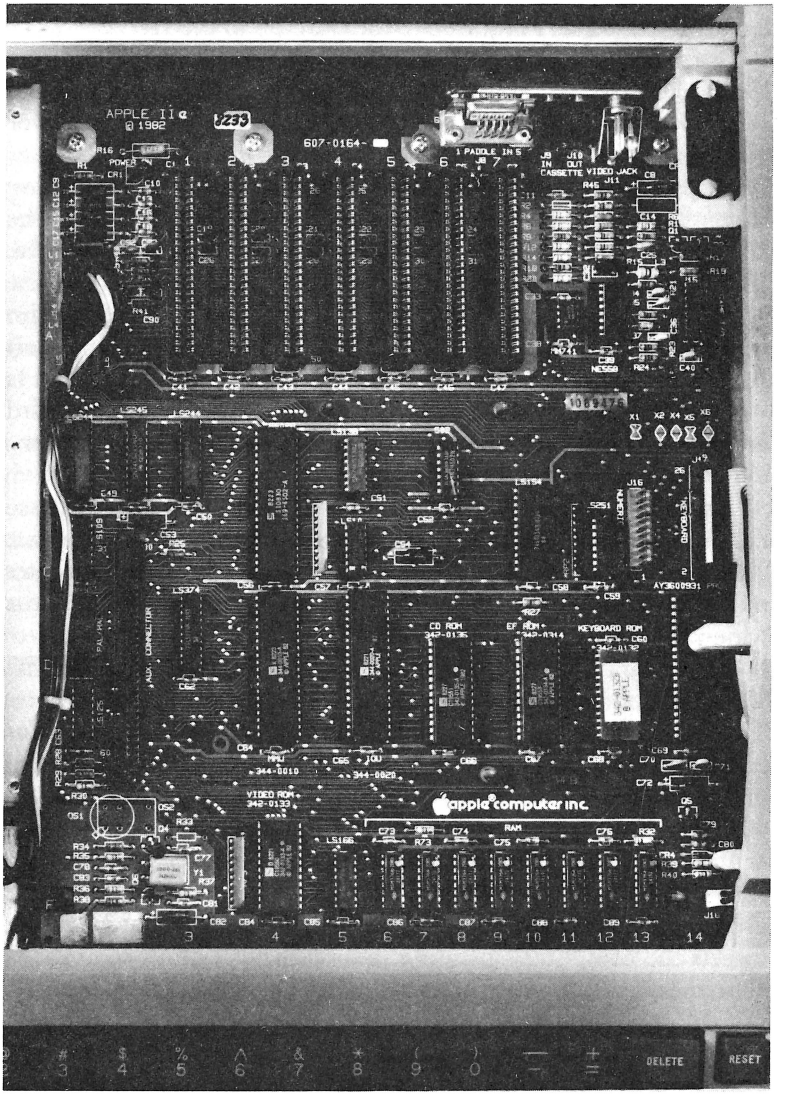


Figure 7.1: The IIe microcircuitry on its "motherboard."

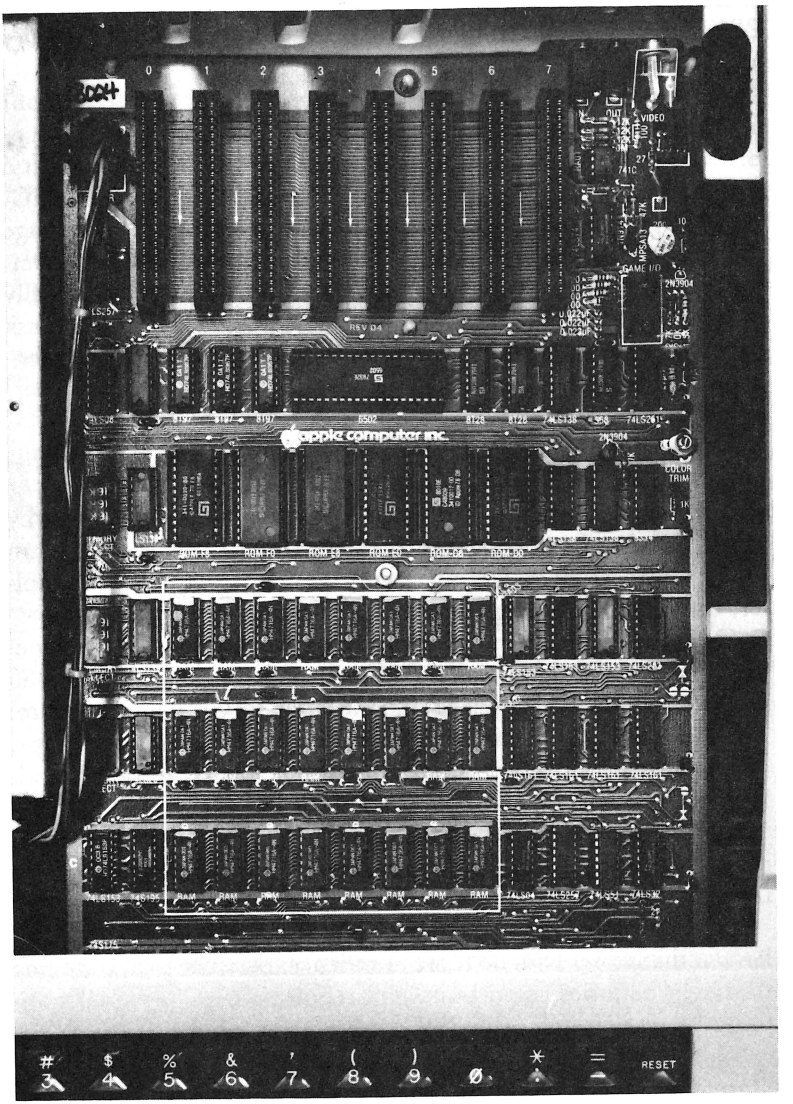


Figure 7.2: The II + microcircuitry on its "motherboard."

machine, and are transferable, provided they don't refer through PEEK, POKE or CALL statements to the few obscure memory locations used differently in each.

The earliest version, simply called the Apple II, took directions through a different variation of the BASIC commands, Integer BASIC, although it, too, could accommodate a language card and so take directions from a second language, like Applesoft BASIC.

A program written in the Apple's numeric machine language can produce different results on different versions, if it uses memory locations that differ between the versions. Some commercially available programs on disks are written in this code. You might ask the salesperson offering such a disk for a guarantee that it's compatible with the version you're using.

EXPANDING THE COMPUTER

Your Apple II wasn't finished at the factory. You can see this by looking under the lid at the empty connector slots there. You can finish construction by adding circuitry cards, like the disk controller, until the machine before you is the one you want.

Other cards—plastic boards packed with electronics—can function in the Apple II in ways of their own. One such card crowds 80 characters of type on a video screen where only 40 would fit before. Another card adds a different processor and its attendant chips, and so grafts a second computer into the Apple II. Another sends information and programs out from the computer through a box to telephone lines and distant receivers and back, or to printers that make the glimmer of a video screen into tangible typed paper. Figure 7.3 shows several of the plug-in cards available for the Apple.

One reason the Apple II computer became so enormously popular was that a user with no more electrical experience than changing flashlight batteries could build and rebuild a truly personal computer by plugging in or pulling out circuitry cards.

For instance, plugging an 80-column card into the solitary auxiliary slot puts a switchable 80-character display option into the computer. That card is one of several links to practical *word processing,* a puffed-up phrase that means nothing more than moving words around in a computer's memory instead of on a sheet of paper. Another link, a set of directions for word-processing commands, comes most conveniently on disk as commercially

available word-processing programs. If the words so processed are to eventually take physical form as ink on paper, a printer will form the end of the linked chain. The link between printer and computer that completes the connection takes the form of an interface card, several varieties of which are available depending on the printer and applications. Known as communications cards, serial interface cards, and parallel interface cards, these circuits act as gateways between printer and computer. Figure 7.4 shows two of the devices these cards can connect to the Apple II, a phone modem coupler and a printer.

Thousands of commercially available programs have been written for the hundreds of thousands of Apples in the world, and they each work through the 6502 processor chip. Another very popular chip, the Z80, has also been well provided with available programs. You can, by plugging in a circuitry card containing the Z80 and its entourage, put a second processor into the Apple II, one that can use the Apple's memory and can take commands from the keyboard or disk. An Apple II so equipped can be switched back and forth between the resident 6502 chip and the Z80 chip. In effect, you are switching between two computers in one box.

The 6502 chip processes commands and information according to directions contained in the Apple II's permanent memory. Another set of directions for controlling processors has become

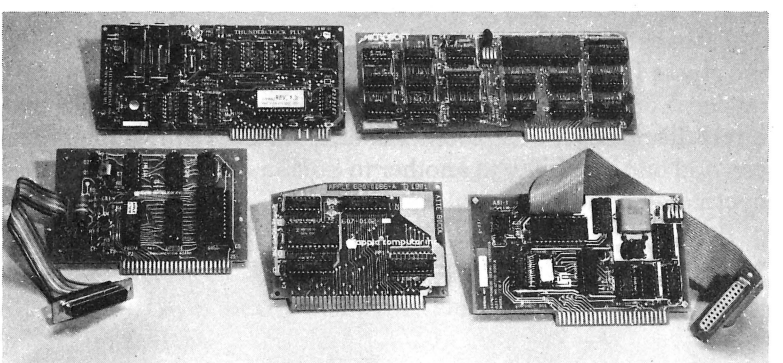


Figure 7.3: Plug-in cards that add circuitry to the Apple II. Clockwise from the top left: an electronic clock, a second processor, a data pathway to outside the computer, an 80-column display circuit, and another data pathway.

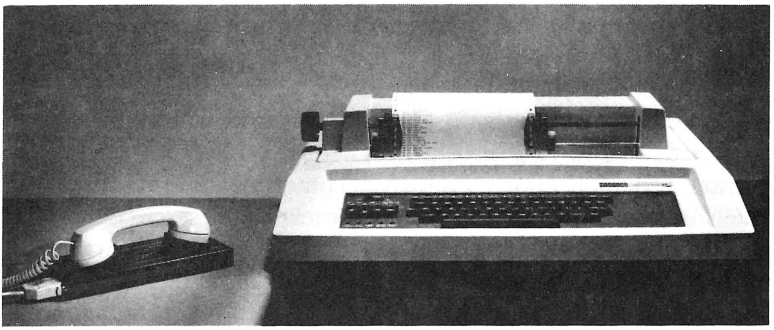


Figure 7.4: A telephone modem and printer that connect to the Apple II.

very popular and is often fed into accepting chips, like the Z80, from a disk. This set is known by the acronym CP/M, and as a system of controlling the processor's operation, has become very popular. In fact, many programs written on different machines using the CP/M operating system are transferable from one type of computer to another with no or few revisions. With a Z80 card installed in the Apple II, you can load this CP/M into the computer.

Finally, most experiences are best shared. If you have the interest, you can open pathways between your computer and other computers and sources of information (even other people!). With a communications card connected to a device used with a telephone, the computer becomes an electronic teletype, linking you to people and data banks in ways that were once the exclusive privilege of large corporations and governments.

Perhaps the most promising feature of a personal computer like the Apple II is its adaptablility to the tastes and preferences of its owner. You need not study the electronics or logic of the chips that direct the machine's capabilities in order to use them. Rather, you can add one capability to another to build a computer system that operates in a most personal way.

And, to that end, using computers in a personal and fulfilling way, best wishes!

APPENDIX A:

SPECIAL PURPOSE COMMANDS

The built-in vocabulary of the Apple II includes commands of more specialized usefulness than those presented in the preceding chapters. Many of these are grouped below according to function. Here an underline indicates where you can include a number or variable in each command.

NUMBER TREATMENT

Mathematical and Trigonometric

ABS (_____): Turns a negative number positive, leaves positive number unchanged.

ATN (_____): Gives the trigonometric value, arctangent, of a number given in radians.

COS (_____): Gives the trigonometric value, cosine, of a number given in radians.

EXP (_____): Gives the value of *e*, 2.7182818, raised to the power of a number.

INT (_____): Gives the value of rounding down to the nearest integer number.

LOG (_____): Gives the logarithm of a number.

RND (_____): Gives a random value between 0 and 1.0, regardless of number.

SGN (_____): Gives the sign of a number: −1 if number is negative, +1 if positive, 0 if 0.

SIN (_____): Gives the trigonometric value, sine, of a number given in radians.

SQR (_____): Gives the square root of a number.

TAN (_____): Gives the trigonometric value, tangent, of a number given in radians.

_____ ^ _____ : Raises the number before it to the power of the number after it.

_____**E**_____ : Multiplies the number before it by 10 to the power of the number after it.

FN _____ (_____): Gives a number according to the mathematical relation established with a DEF command. See DEF FN. FN Z(A), FN Z(B) and FN Z(5), for example, each call on the same relation set as FN Z(_____).

DEF FN _____ (_____) = : Establishes a mathematical equivalence between the variable in parentheses and the mathematic relation after the equal sign. For example, DEF FN X(A) = A + B establishes a function FN X(_____), which adds the value of B to the value or variable in parentheses.

Comparisons

Comparison operators give responses that correspond to the truth or falsehood of the statement made with them. When the statement is true, a value of 1 is given; when false, a value of 0.

_____>_____ : Gives a value of 1 if the number before it is greater than the number after it, 0 if otherwise.

_____<_____ : Gives a value of 1 if the number before it is less than the number after it, 0 if otherwise.

_____ =_____ : Gives a value of 1 if the number before it is the same as the number after it, 0 if otherwise.

_____<>_____ or _____><_____ or _____#_____ : Gives a value of 1 if the number before it is not the same as the number after it, 0 if it is.

_____>=_____ or _____ =>_____ : Gives a value of 1 if the number before it is greater than or equal to the number after it, 0 if it is less.

_____<=_____ or _____ =<_____ : Gives a value of 1 if the number before it is less than or equal to the number after it, 0 if it's greater.

Logic (Boolean)

____AND____: Gives a value of 1 if the number or relation before it and the one after it are both non-zero, 0 otherwise.

____OR____ : Gives a value of 1 if either the number or relation before it or the one after it is non-zero, 0 if they are both 0.

NOT____ : Gives a value of 1 if the number after it is 0, a value of 0 otherwise.

Other

ASC (" ____ ") : Gives a code number, called the ASCII code, for the first character of the string of characters.

STR$ (____) : Directs the computer to treat a number as a string character.

VAL(____$) : Directs the computer to treat a numeric string character as a number.

COMPUTER STATUS

SCRN (____,____) : Gives the color-code number for a column, row location in 40-by-40 graphics.

POS (0) : Gives the value of the location of the cursor's column position.

COMPUTER CONTROL

HGR2: Switches the computer to display 280-by-192 graphics and respond to the same commands as HGR switching. The resulting display is like 280-by-160 graphics, but with the four lines of text window replaced by the graph screen.

CLEAR: Replaces all variables in a program with the value 0.

IN # ____: Directs computer to the slot from which data will be received.

APPENDIX B:

JARGON PHRASE GUIDE

Jargon is the curse of the specialized, and every specialty has its jargon. You will encounter computer jargon among enthusiasts, salespeople, and programmers. Jargon speakers, who are often more knowledgeable, can provide useful information not obtainable from speakers of the mother tongue. The phrases that follow can help in making them intelligible.

Applications program: A set of commands, usually available on disk, that directs the computer in performing some task whose results appear in a recognizable form outside the computer's memory.

Back-up: Not directions given to a truck driver, but a copy kept in case the one in use should fail. A back-up copy of a disk can be kept, or a back-up copy of a file or program can be kept in another form or on the same disk.

Bit: Not the past tense of bite, but the smallest piece of information the computer can handle. A bit always has a value of either 0 or 1.

Boot a Disk: Not an act of aggravation, but the act of sending directions that operate a disk system into the computer from a disk in a drive. (You boot the disk before sending disk commands to the computer.)

Boot DOS: See *Boot a Disk.*

Buffer: Won't shine shoes, but, as a specially controlled memory, will hold data or commands being moved between the computer and other electronic devices. It's useful when the rate of flow out of one is greater than the rate into the other, like a reservoir collecting a river's waters faster than it lets them through flood gates.

Bug: Not a crawling six-legged creature, but any unplanned response that interferes with the expected operation of a machine. The term usually refers to an unplanned command problem within a computer program.

Byte: Not Old English for the use of teeth, but the basic package of information sent through the computer circuits, in which each of eight bits has a value of either 0 or 1.

COBOL: A programming language in widespread use in business since before the dawning of the age of smaller, personal computers.

Copy-protected: Characteristic of a program on a disk that includes directions to intentionally thwart efforts to copy that program onto another disk.

CPU: Central Processing Unit, the processor in the computer.

Crash: In a program run, what happens when the computer stops acting on program commands suddenly, in an unplanned way, usually because of directions the computer can't follow.

Data-base Manager: Not a baseball coach, but a program that organizes information according to a preset plan.

Default (dear Brutus, lies not in our stars but in ourselves): That value chosen automatically by the computer when a choice is not made by the computer user.

Enter: To send data or instructions into the computer, usually typed from the keyboard, and followed by a subsequent press of the RETURN key.

Error: Any command or data given to the computer that it is incapable of acting on.

Escape Sequence: The use of keyboard typing to stop the running of a program or part of a program.

Fatal Crash: The kind of *crash* in which program commands or valuable information is lost from computer memory.

Firmware: Not a corset, but those directions stored in built-in memory that can't be changed from the keyboard. (Compare with *hardware* and *software.*)

Format: To etch a magnetic pattern onto a disk, so that programs and data can be stored on it.

FORTRAN: A computer programming language used by scientists and engineers since the days of punch cards.

Function: A command that manipulates values, often in some unseen way. The commands +, −, *, and / represent arithmetic functions.

Hardware: Not nuts and bolts, but the physical parts of a computer system, as opposed to the directions and values stored in it. (Compare with *firmware* and *software.*)

Hex: Not a witch's curse, but an abbreviation for hexadecimal, a way of representing numbers by a hybrid of the first six (hence hex) letters of the alphabet and the ten (hence decimal) numbers.

High-level Language: Not the King's English, but a set of commands that direct the computer in relatively complicated ways to perform a complex task with a single command. BASIC, both the Integer and Applesoft versions, is a high-level language.

Housekeeping: Not dusting and mopping, but the chores done by the computer user, or the computer itself, to operate a system or a program.

Initialize: To format a disk and add an initial program of commands that will be automatically run by the computer when the disk is first scanned.

Instruction: Any command or direction given to the computer.

Interface: As a noun, the circuitry that serves as electronic connection to the computer, or a method of commands that connects different logic systems. As a verb, to connect.

I/O: The flow of data or commands to and from the computer, from Input/Output.

K: When referring to the number of memory locations, an amount roughly equal to a thousand. Its exact value is 1024. 64K is equal to 65,536.

Logo: A programming language designed for ease of use and learning, often a child's introduction to computers.

Loop: A series of commands through which a computer is directed repeatedly.

Low-level Language: Not locker-room talk, but a set of commands that direct the computer in relatively unsophisticated ways, requiring several commands to direct a complex task.

Machine language: The codes for memory locations particular to any computer.

Menu: Not a selection of entrees at a restaurant, but a list of choices given in a program, from which its user selects different operations. Programs using them are called *menu-driven.*

Microcomputer: A computer built around a microprocessor.

Microprocessor: The electronic circuitry chip encased in plastic, and controlling the rest of the computer.

Mini-Floppy Diskette: The formal name for a disk, 5-1/4 inches across and flexible, used with the Apple II and most other personal computers.

MMU: (Memory Management Unit) The part of the computer that organizes the flow of information in bits and bytes into different parts of memory according to preset pattern.

Modem: A device that couples a computer to telephone lines.

Parallel: Travelling concurrently, as in the way data or command impluses can be sent from a computer through a set of parallel wires. (Contrast with *serial.*)

Pascal: A computer programming language that is particularly popular in academic circles.

PC Board: (Printed-Circuit Board) A hard plastic or fiber board on which strips of bonded foil serve as electrical pathways in place of wires, and on which electronics are seated.

Peripheral: Any device attached to the computer controlled by, or controlling, it.

Pixel: Not a mincing dwarf, but the smallest dot a computer can control on a video screen.

Powerful Language: Not epithets, but a set of commands, each of which can give directions that only several commands of a "weaker" language could. (See *High-level Language.*)

Powerful Program: A program that does a lot with little prompting or control by its user.

RAM: Not a horned sheep, but an acronym for Random-Access Memory, the computer memory locations whose values can be changed from the keyboard.

Read: To draw information from an encoded form, as in to read from a disk.

Return a Value: What the computer does when it displays, on screen or printer, numbers or characters in response to commands.

ROM: An acronym for Read-Only Memory, the computer memory locations with values fixed in manufacture. It provides all built-in directions for the computer.

Serial: One after another, as in the way data or command impulses can be sent from a computer through a single wire. (Contrast with *parallel.*)

Software: Not lingerie, but the directions, such as those available as programs on disk, that can be added to and removed from the computer system, as opposed to physical, fixed parts. (Compare with *firmware* and *hardware.*)

Spreadsheet: Not bed linen hanging to dry, but a visual system of organizing data, often business or financial, according to rows and columns. Several such systems are available as programs on disks.

Utility: Not the gas company, but a program that directs the computer in performing some internal task, like moving data about its memory.

Write: To put information in a form in which it can be stored and then retrieved, as in to write to a disk.

Write-enable Notch: Not the secret to great literature, but a section of the disk envelope that, when uncovered, switches the disk drive so that it can add or alter information on the disk.

Write-protected: The physical feature of a disk that prevents it from having information added or altered. The absence of, or taping over of, a notch on the disk envelope results in write-protection.

APPENDIX C:

BOOKS OF INTEREST

Each of the following books provides a useful extension of the introductory discusssions presented in this guide.

Disk Use:

The DOS Manual Apple II Disk Operating System. Cupertino, Calif.: Apple Computer, Inc., 1981.

The authoritative source document from the manufacturer on its disk operating system (DOS 3.3), including all disk commands and details of how to use them. A complete reference if you plan to learn disk-system use inside and out.

Beneath Apple DOS. Worth, Don and Lechner, Pieter. Reseda, Calif.: Quality Software, 1981.

A descriptive account of the disk operating system, which makes no attempt to teach commands, but rather explains the details of this system's operation.

Programming:

Basic Exercises for the Apple. Lamoitier, J.P. Berkeley, Calif.: Sybex Inc., 1982.

An example-by-example approach to Applesoft BASIC programming techniques, covering a wide range of applications for the reader who is not intimidated by mathematics.

Summary Reference:

Programmmer's Guide to the Apple II. Granite City, Ill.: Computer Station, 1978.

Handy lists of Applesoft BASIC commands, disk commands, and PEEK, POKE, and CALL addresses for the reader who has already learned how to use them.

Machine Language:

Apple Machine Language. Inman, Don and Inman, Kurt. Reston, Va.: Reston Publishing, 1981.

A book that provides a sound and unrushed explanation for the person prepared to use numbers and hexadecimal codes to direct the computer, and to make full use of PEEK, POKE, and CALL commands.

The Computer Itself:

Apple IIe Reference Manual. Cupertino, Calif.: Apple Computer, Inc., 1983.

The authoritative book from the manufacturer that describes the logical and electrical organization of the machine, from memory addresses to physical components.

INDEX

Adding words and strings, 63
Altering disk programs, 106–110
Animation, 59
Apple II computers, differences among, 128–132
Applesoft BASIC commands, 93, 94, 123, 127, 129
Arithmetic, 12
Arrow keys, 6, 7
Assignment commands, 34
Automatic decision making 37–41
Automatic disk operation, 92

BASIC language commands, 93, 127
Beeps, 16
Binary-language program, 95
BLOAD, 96
BRUN, 95

CALL, 127
CATALOG, 93
CATALOG listing, 93, 94, 95, 99
Character displays, 10, 11, 12
Clearing the screen, 9
CLOSE, 104
COLOR = , 21
Commands, 8
Computer memory, 27, 123
Computer simulation, 34–37
Computerized conversation, 77
Conditional command, 37
Connecting a second drive, 87
CONT, 47, 48
CONTROL-C key command, 47
CONTROL-G keypress, 16
CONTROL/HOLLOW-APPLE/RESET, 91
CONTROL-RESET, 78, 79
Counting word & string lengths, 63–64
CP/M operating system, 134
Creative plans, 111
Cursor, 5, 19
Cutting words and strings, 64–68

DATA, 68–74
Data banks, 134
Decimal code, 124
DEL, 49
DELETE, 99, 100
DIM, 73, 74
Disk(s), 84, 85, 88
Disk directions, 91, 92, 93, 101, 102
Disk drive, 84–88
Disk programs, linking, 117–120
Disk system, 83–88, 90, 91
DOS disk, 91, 92, 93, 97, 117
Drawing arrows, 54–61
Drawing with paddles, 82
Duplicating a disk, 97

80-column card, 132, 133
Electronic combination lock, 81
Electronic sensing, 79–82
Empty loops, 60
END, 47
Error, 7
Error detection, 53
ESC key, 7
EXEC, 120, 121

FLASH, 12, 13
FOR-NEXT loops, 54–61
Formatting a disk, 101–102
FP, 94

GET, 78, 79
GOSUB, 111, 112, 113, 114
GOTO, 50, 51
GR, 20
Graphics, high resolution, 23–26

Graphics, low resolution, 20-23
Greeting disk program, 94, 98, 101

HCOLOR =, 24
Hexadecimal code, 124
HGR, 24
HLIN, 22
HOME, 9, 10
HPLOT, 24
HTAB(), 19

IF-THEN, 37, 38, 39, 40, 41
Information, adding to programs, 75-82
Information files, 103-110
Information handling, 68-74
INIT, 101, 102
Initializing a disk, 101-102
INPUT, 75, 76, 77
Input loop, stopping, 78-79
INT, 95
Integer BASIC, 93, 132
Interactive programs, 77
Interface cards, 132-133
Internal operation, 14
INVERSE, 11
Inverse video, 11

Jargon, 139-144

Keyboard, 6, 128

Language card, 132
Languages, switching between, 94-95
LEFT$(), 64
LEN(), 63
LIST, 28
LOAD, 96, 100
LOCK, 101
Loops, 50

Machine-language commands, 127
Main program, 117
MAKE TEXT program, 117, 119, 120
Mathematic commands, 135-136
Mathematical models, 34
Memory addresses, 27, 129
Microcircuitry, 128-131
MID$(), 66, 67
Modem, 133
Modular programs, 114
MON, 105
Moving programs between disks, 100

Naming a disk program, 98
Nested loops, 56-59
NEW, 31
NORMAL, 12, 13
NOTRACE, 45, 46
Numbering statements, 27-32
Numeric comparison commands, 136-137
Numeric logic commands, 137

ON-GOSUB, 115, 116
ON-GOTO, 51, 52
ONERR-GOTO, 53
OPEN, 104

Paddle use, 79-82
PDL(), 79, 80, 81, 82
PEEK(), 126, 127
PLOT, 21
POKE, 125, 126, 127
PR#6, 92
PRINT, 14, 15, 16, 17
Processor, 123, 125, 129, 133, 134
Program building, 117
Program memory, 31
Programs, notes within, 43-45
Program parts, 111
Programs that run other programs, 118-119
Prompt, 6, 94

READ, 68-74, 105

Read-only, 89
REM, 43, 44, 45
Removing programs, 31
Removing program statements, 30
Removing sections of programs, 49
RENAME, 99, 100
Repetitions, 54-61
Replacing program statements, 30
Reserving space for variables, 73-74
Resetting the computer, 7
Restarting a program, 47
RESUME, 53
RETURN, 111, 112, 114
Return key, 6
Reusing variables, 71-73
RIGHT$(), 66
RUN, 28, 96, 121

SAVE, 98, 100
Screen displays, 20-26
Screen grid, 16
Screen window, 124
Sending commands, 9
Series of commands, 18
Sounds, 16
SPEED, 17
SPEED =, 18
Speed of printing, 17-18
STOP, 46
Stopping a program, 46-49
Stopping disk programs, 119
Strings, 61-68
Subroutines, 111-117
Subscripted variables, 71-74
Substitutions, 33-37, 69
Symmetric patterns, 22
Syntax error, 7
System master, 91

TAB(), 16
Television, 4
TEXT, 23
Text display, 20
Text files, 103-110
Text window, 124-127
Time delays, 60
TRACE, 45, 46
Translator, 7, 123, 125
Trigonometric commands, 135-136

UNLOCK, 101
Using disk programs, 90, 94-97

Variable, 34
Variables, 33-37
Video monitor, 3
Video screen, 9
VLIN, 22
Vocabulary, 7, 61
Vocabulary of disk operating directions, 88
VTAB(), 19
Vulnerability of disks, 89-90

Word manipulations, 61-68
Word processing, 132
WRITE, 104
" Write-enable" notch, 88, 89

Z80 card, 134

A Selection from the SYBEX Library

YOUR FIRST APPLE II® PROGRAM
by Rodnay Zaks 150 pp., illustr., Ref. 0-136

THE APPLE II® BASIC HANDBOOK
by Douglas Hergert 144 pp., illustr., Ref. 0-115

APPLE II® BASIC PROGRAMS IN MINUTES
by Stanley R. Trost 150 pp.,illustr., Ref. 0-121

BASIC EXERCISES FOR THE APPLE®
by J. P. Lamoitier 250 pp., 90 illustr., Ref. 0-084

THE APPLE® CONNECTION
by James W. Coffron 264 pp., 120 illustr., Ref. 0-085

PROGRAMMING THE 6502
by Rodnay Zaks 386 pp., 160 illustr., Ref. 0-046

APPLE® PASCAL GAMES
by Douglas Hergert and Joseph T. Kalash 372 pp., 40 illustr., Ref. 0-074

MASTERING VISICALC®
by Douglas Hergert 217 pp., 140 illustr., Ref. 0-090

DOING BUSINESS WITH VISICALC®
by Stanley R. Trost 260 pp., Ref. 0-086

VISICALC® FOR SCIENCE AND ENGINEERING
by Stanley R. Trost & Charles Pomernacki 225 pp., illustr., Ref. 0-096

INTRODUCTION TO WORD PROCESSING
by Hal Glatzer 205 pp., 140 illustr., Ref. 0-076

COMPUTER POWER FOR YOUR LAW OFFICE
by Daniel Remer 225 pp., Ref. 0-109

DON'T (or How to Care for Your Computer)
by Rodnay Zaks 214 pp., 100 illustr., Ref. 0-065

YOUR FIRST COMPUTER
by Rodnay Zaks 258 pp., 150 illustr., Ref. 0-045

INTERNATIONAL MICROCOMPUTER DICTIONARY
120 pp., Ref. 0-067

For a complete catalog of our publications

2344 Sixth Street, Berkeley, CA 94710
Tel: (415)848-8233, Telex: 336311

SYBEX®

FOR THE NEW APPLE IIe AND APPLE II+

THE EASY GUIDE TO YOUR APPLE II

This is the book that will show you how to use your Apple IIe or Apple II+ in a matter of hours. Right away you will become familiar with the keyboard, video screen, and disk drives.

You can learn how easy it is to write your own BASIC programs. Or, if you prefer, you can skip programming and learn how to get started with the enormous amount of commercially available software.

You will quickly learn to:

- Create video graphics
- Use your disk drives
- Make forecasts and simulations
- Customize pre-recorded programs to your needs
- Expand your system with useful accessories
- And much more

Everything you need to know to get started with your Apple personal computer is revealed in a friendly, jargon-free style.

ABOUT THE AUTHOR:

Joseph Kascmer is a professional writer whose works include articles and instructional materials in health sciences, anthropology, politics, regional history, and computers in business and education.

Kascmer

The Easy Guide to Your Apple II

SYBEX

0-122